个性化你的阅读

编程狂人

Programming Madman

NO.13

推酷

关于推酷

推酷是专注于 IT 圈的个性化阅读社区。我们利用智能算法，从海量文章资讯中挖掘出高质量的内容，并通过分析用户的阅读偏好，准实时推荐给你最感兴趣的内容。我们推荐的内容包含科技、创业、设计、技术、营销等多方面内容，满足你日常的专业阅读需要。我们针对 IT 人还做了个活动频道，它聚合了 IT 圈最新最全的线上线下活动，使 IT 人能更方便地找到感兴趣的活动信息。

关于周刊

推酷周刊是专为 IT 人打造的行业技术周刊，目前推出的《编程狂人》是献给 广大的程序员们。 我们利用技术挖掘出那些高质量的文章，并通过人工加以筛 选整理出来。 每期的周刊一般会周一的某个时间点发布。

联系我们

tuicool2012　　164644910　　推酷网

下载 APP

Android版本

立即下载 Android

iPhone版本

免费下载就在 App Store

2014/02/24/第十三期

目录

技术纵横

后端架构

程序人生

正文

[业界新闻]

2014 年最值得学习的编程语言

经过数据分析和研究 Jobs Tractor 的 45000 个开发人员招聘职位数据，我们得到了如下的结果：

Java 8,731

PHP 8,238

Objective-C 5,859

Java for Android 4,312

SQL 3,553

JavaScript 3,154

Ruby 2,937

C# 2,549

Python 1,587

C++ 1,273

C 685

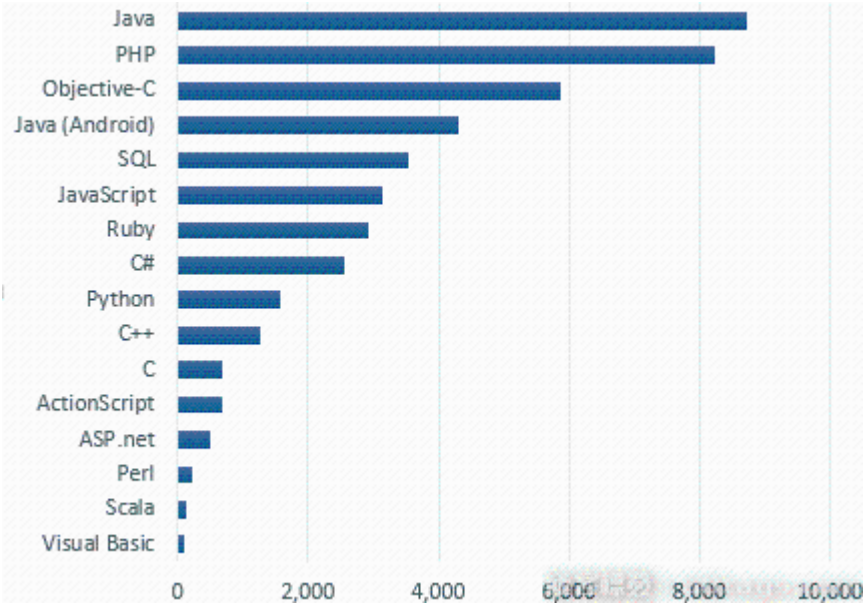
ActionScript 674

ASP.net 492

Perl 224

Scala 143

Visual Basic 92



自上一年，主要的变化如下：

PHP 和 Java 换了位置，但是仍旧是高居不下

Java 的 Android 已经取代了 SQL 并且接近 Objective-C

Javascript 已经超过了 Ruby

Python 提高了 2 位

ActionScript 下降了 2 位

以上数据来自 Twitter，由于地域或者其它原因可能导致结果的偏差。

近来的一个 Lynda 的在线调查提供了如下一个流行度的排名：

Java

C

C++

C#

Objective-C

PHP

Python

Ruby

JavaScript

SQL

当然，这里也有一些疑问，网站的主要用户是相对新的开发人员。

工业分析 RedMonk 采用了不同的方式来生成了最受欢迎的编程语言，主要通过了 Github 上的项目和 StackOverflow 上的提问来生成。

JavaScript

Java

PHP

C#

Python

C++

Ruby

C

Objective-C

CSS

不要太书面化的看待这个结果，一个语言包含更多问题可能会得到更高的分数。虽然类似 SASS，LESS 和 Stylus 之类的预处理器的出现使得 CSS 也出现在列表中并不奇怪，因为现在它看起来更想一个编程语言。当然，如果说 CSS 是一个编程语言，那么 HTML 和 SQL 呢？

需要建议 - 不需要分析和数据

永远不要使用分析作为学习语言的唯一基础

很少开发人员是为了经济上的原因来学习编程的，这里有很多其它赚钱的方式... 例如成为“数字市场分析师”，或者“SEO 专员”

选择一个流行的编程语言意味着你需要说服其它语言。学习 Fortran 可能不会非常流行但是你将会发现维护 10 几年的老系统还是很有“钱途”的。幸运的是，我们可以提供几个比较常用的方向来帮助你选择你的学习目标。

技术的起起伏伏

所有的语言在流行度上都是起起伏伏的，不同的时间段也将不一样。考虑一下 ActionScript。Flash 开发正在走下坡路，当然个人怀疑很多的 Actionscript 项目

也需要维护。同样的问题也出现在 Perl，COBOL 和 VB6，虽然他们拥有超长的生命周期。

如果你考虑这些的话，你需要避免使用平台有限的语言，例如，Actionscript，VB6 和 Objective-C。然而，Objective-C 主要使用在 iOS 系统和 API，但是目前来说对于 ios 的 app 开发，工作也不少。

老手开发人员的选择

如果你已经熟练掌握了 1 到 2 门的语言，选择就更简单了：选择你感兴趣的（知性或者经济上来考虑）

这里有一些明显的机会，例如：

ActionScript 基于 ECMAScript(浏览器中的执行就是 Javascript)，Flash 开发人员来说更靠近 HTML5 的技术。

C++，Java，C#，Object-c 甚至是 PHP 都非常类似，因此你可能需要经常在它们之间换换

如果你使用 VS，.net 来开发 windows 桌面应用，那么应该选择微软平台

尽管这些，不要害怕学习更多新的技术。JS 看起来类似 Java 和其它 C 风格的语言，但是很多开发人员开始觉得很痛苦，因为基础不太一样。坚持一下你就看到了 Javascript 能够提供的强大功能。

新手开发人员的选择

对于那些使用 8 位家用机时代的开发人员来说并没有什么可以选择，在有足够信心后可以学习 C 或者汇编之前先学习 Basic 吧。web 开发的黎明相对来说更简单；你可以学习 HTMl 及其服务器端的语言例如，Perl。你的开发技术会随着 HTML 及其其它例如 CSS，PHP，Javascript，ASP 和.Net 的发展而慢慢提升

个人并不妒忌 2014 年才开始学习编程开发的人；过多选择会让人迷糊，那么你如何开始？

暂时来说，个人推荐 JavaScript。这个语言可以在很多场景下使用，发展的很快并且拥有很多的在线的资源。学习 Javascript 可以帮助你避免其他开发人员的开发经验。唯一比较让人犹豫的是 JS 处于浏览器环境。JS 可能要求额外的一些客户端知识，例如，HTML，CSS 和跨浏览器兼容，即使你开发 node.js 的服务器端代码开发。

另外一个选择，你应用考虑类似 Ruby 和 Python 的编程语言，相对来说学习更简单，也没有环境和遗留系统的问题。然而，他们提供了比较少的资源，并且来自 C 风格的语法，这个可能成为你的最后目标。

最好的建议是：不要再阅读类似的建议类文章。

使用软件工具来定位并且解决问题。使用 Autohotkey 来自动化任务或者使用 Macro 来编写 Spreadsheet 的计算公式。这些知识都可以提供足够的储备帮助你学习更大更复杂的编程任务。

提问：如果你进来打算学习编程，你想选择什么语言呢？是否帮助或者阻碍你的学习？你推荐什么给新的开发人员呢？

原文链接：http://www.linuxeden.com/html/news/20140217/148500.html

Elasticsearch 1.0.0 发布

Elasticsearch 发布了以其自身命名的开源分析工具的 1.0.0 版本。Elasticsearch 是一款分布式搜索引擎，支持在大数据环境中进行实时数据分析。它基于 Apache Lucene 文本搜索引擎，内部功能通过 ReST API 暴露给外部。除了通过 HTTP 直接访问 Elasticsearch，还可以通过支持 Java、JavaScript、Python 及更多语言的客户端库来访问。它也支持集成 Apache Hadoop 环境。Elasticsearch 在有些处理海量数据的公司中已经有所应用，如 GitHub、Foursquare 和 SoundCloud 等。

Elasticsearch 的基本特性主要围绕可伸缩性、高可用性和实时分析。进入搜索引擎的数据会立即建立索引，在集群中复制，并为分析做好准备。

可伸缩性：Elasticsearch 是为在集群环境中工作而设计的。一个节点一启动，就会自动寻找网络中的其他节点并连接过去。索引以分片方式组织，分布在集群中。因此搜索索引是分布式操作，会在所有集群节点上并行运行。如果需要更好

的性能，只需要将额外的节点加入集群，分片会自动识别。

可用性：数据库分片不仅用于水平伸缩，也有可用性考虑。对于每个分片，都有一个保存在不同的集群节点上的复制分片，所以如果一个节点当掉，并不会丢失数据。Elasticsearch 会探测到故障节点，将其集群中移除。在故障节点移除之后，考虑到伸缩性和弹性，分片会重组，以便优化。

为支持整个集群重启，Elasticsearch 所需要的所有元数据能够持久化到各种存储类型上。数据可以借助所谓的网关来存储，网关目前支持本地存储和共享的文件系统。

实时：Elasticsearch 是无模式的，而且支持索引任意的 JSON 文档。它会分析文档的结构，甚至还能自动探测某些数据类型，如时间戳。默认情况下，文档中的所有字段都会被索引，而且是可以搜索的。除了简单的全文搜索之外，分面（facets，提供聚合的分析函数，如日期范围、距离、柱状图等）和度量指标（metrics，如求和、平均和统计等）可以直接应用于索引。

Elasticsearch 1.0.0 的新特性

1.0.0 版本对 API 进行了很多修改，并带来了很多功能增强，使 Elasticsearch 用起来更为直观和高效了。功能增强包括备份和恢复索引、分析数据并使 Elasticsearch 更有弹性的新方法：

快照/恢复：新版本提供了一个简单的 API，用来生成整个集群的快照，以创建备份。Elasticsearch 集群的状态——包括元数据和索引——可以保存在快照仓库中。仓库通常放在共享的文件系统中，而且可以保存任意数量的快照。如果发生了内置的容错和弹性机制无法处理的问题，集群可以根据仓库中的任何快照重建。

聚集（Aggregation）：相对于之前版本中已有的分面，聚集为分析现有数据提供了更强大的功能。分面仅为分析功能提供了少量结果（比如，特定距离内的商店数），聚集则会保存某次查询实际找到的文档，并支持将生成的文档集作为新查询的输入（比如，特定距离内所有商店的季度平均销售额）。

断路开关（Circuit Breaker）：系统将添加断路开关，以阻止操作或运行时错误对搜索索引造成严重的不利影响。Elasticsearch 1.0.0 添加的第一重保护是监控空余内存，并评估搜索或分析操作所需要的内存量。如果某个操作需要的内存超出了可用内存，就阻塞该操作，这样就不会导致 OutOfMemory 异常了。未来的

版本中将实现更多断路开关。

Elasticsearch 使用修改主版本号的方式来整理现有的 API，它也接受不向后兼容的修改。在升级到 1.0.0 版本之前，用户应该备份所有数据，并阅读所有破坏性修改的列表。

Elasticsearch 还提供了用于处理数据获取和分析的额外工具。连同 Logstash 和 Kibana，Elasticsearch 还创建了 ELK-stack 来分析日志文件和其他与时间相关的信息源，并以不同的方式对这些数据进行分析和可视化。

也可以通过 Elasticsearch 的商业分支购买专业支持。

原文链接：http://www.infoq.com/cn/news/2014/02/elasticsearch_1.0.0_released

15 个有用的项目管理工具

在如今快节奏的商业世界中，能够通过规划、组织来管理项目，管理资源池并对开发资源完成评估可以说是一项艰巨的任务及责任，其贯穿于个人或团队并决定项目最终期限。

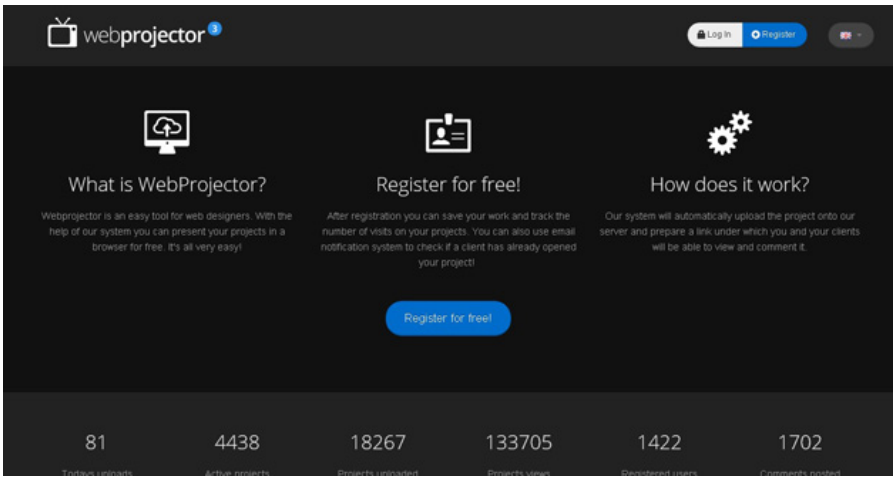
目前有许多基于 PC 的项目管理软件，它们的存在能够帮助减轻负担，并且以它们方式进入到了几乎所有业务类型中。然而，早期时，项目管理软件只能运行在大型计算机并用于大型工程项目中。这些早期的系统在作用及生产力上是相当有限的，以今天的标准看来管理和利用它们非常困难。

在这篇文章中，我们收集了一堆这样的项目管理工具，它们是能够用来处理任何类型的项目协作的工具，应用于运行和维护涉及到你的业务和企业的最困难任务，即使是对最初级的用户都能有所帮助.

高兴吧，那就让我们来了解一下那些项目管理工具适合你，而如果我们漏掉一些好工具 – 请在留言中告诉我们那些工具对你帮助最大，或者那些工具你现

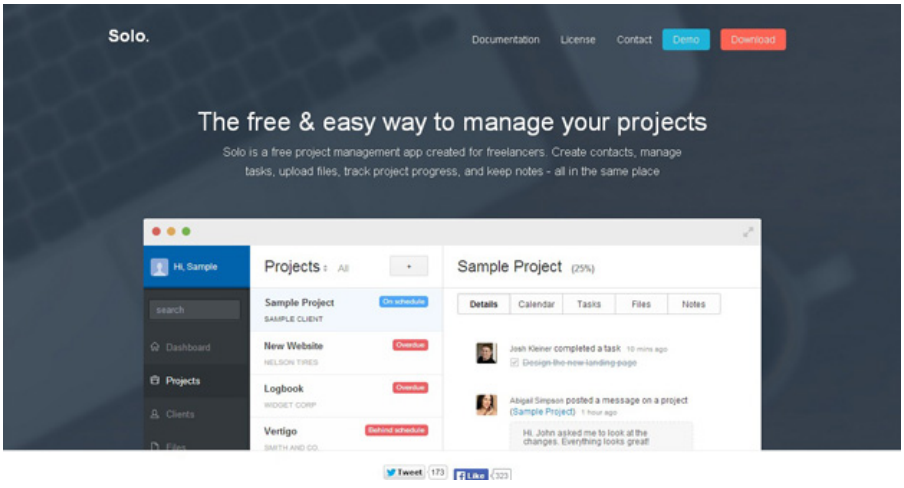
在正在使用.

1. WebProjector



WebProjector 是一个只为此目的而构建的免费的基于 web 的工具. 我们可以简单的通过拖拽上传图片，通过 e-mail 发送链接,而图片会按照我们定好的顺序显示. 这个工具也会在它们被查看或者被多次查看的时候通知你，它还可以将我们的设计封装到手机，平板或者桌面框架这些更加现实的界面中. 另外，对每一个设计进行评论也是可以的.

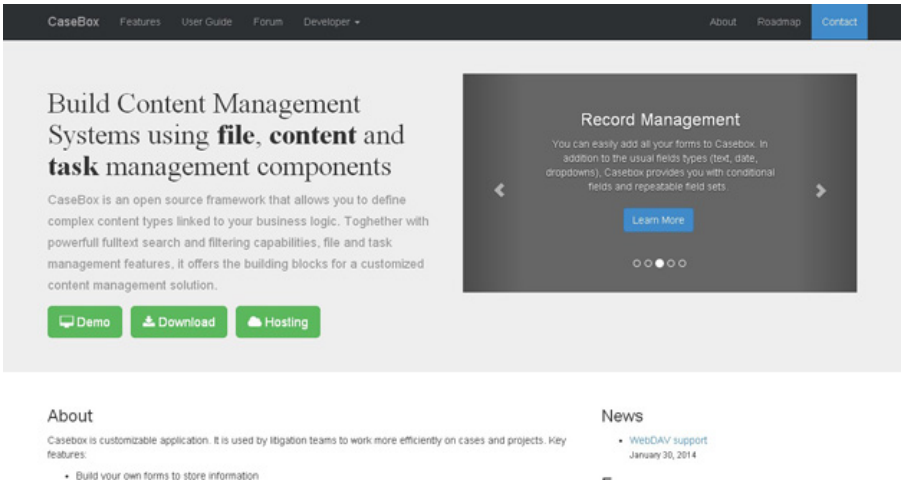
2. Solo



Solo 是一个为自由职业者创建的项目管理应用.它是自托管的,由 PHP 写成并用 MySQL 存储数据。它有一个漂亮的界面，一个用户(限制只能有一个用户)可以创建任意数量的项目+任务并且和客户绑定。任务可以被很容易的搜索，通过拖拽重排序并且可以添加附件。文件上传工具也非常强大，它可以同时上传多个文件并且内置文件预览。自带的项目日历可以快速查看任务，一个活动的流动图展

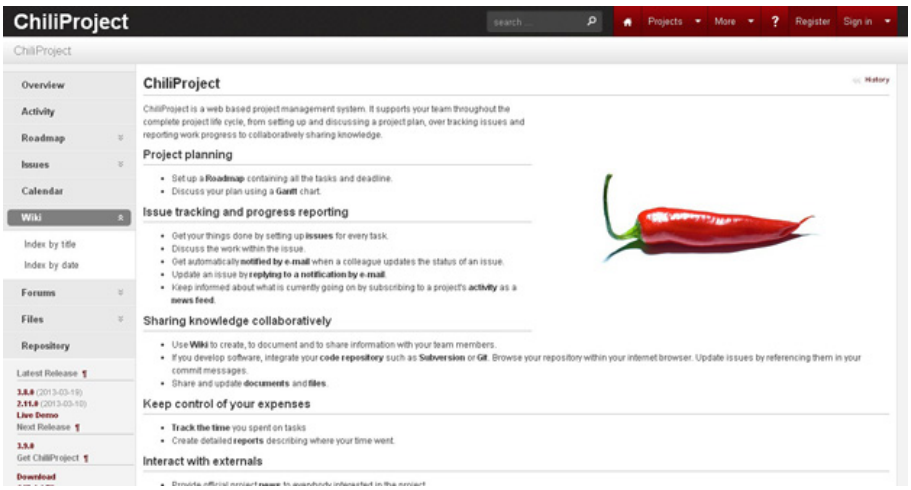
示了你所有的活动踪迹，并且有一个计时器帮助你跟踪你在任务/项目/客户上面所花的时间。

3. Casebox



CaseBox 是一个开源的由 PHP/MySQL 驱动的用于存储和管理记录、任务和文件的 web 应用。它有一个类似于桌面应用的界面并且我们可以在上面建立无线层级的目录并且按照自己喜欢的结构形式存储资料。可以创建带时限的任务并把它们分派给用户而且可以很容易的追踪到进度情况。我们可以在里面储存用户信息和任意数量的文件。文件的内容会被索引所以定位一个 PDF/Word 等文件和它的内容是十分快捷的。

4. Chili Project



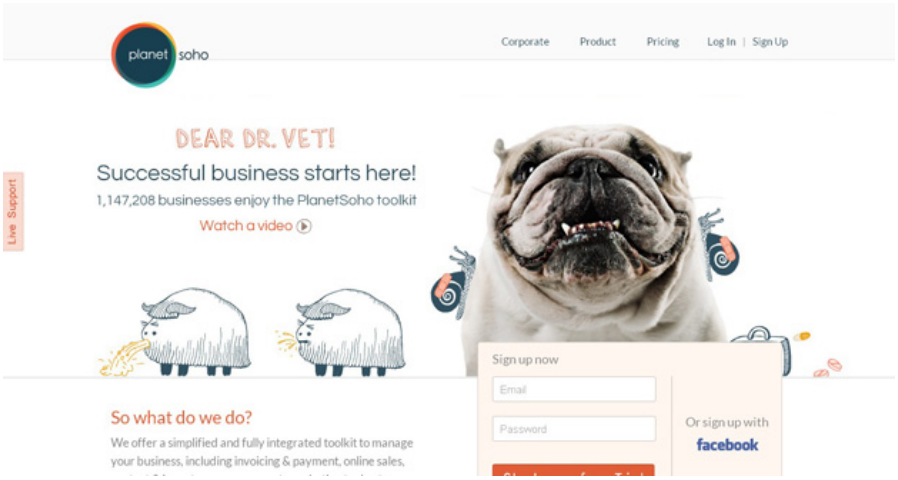
ChiliProject 是一个基于 web 的项目管理系统。它贯穿了你们团队整个的项目生命周期，从立项讨论项目计划到跟踪问题和报告工作来共享知识。ChiliProject 通过 Email 来通知项目成员发生了什么，进一步的通知会通过一个简

单的订阅项目活动 feed 存在。为了记录下细节/知识，它自带了一个 wiki，另外，有一个论坛用于和用户而不只是团队成员讨论问题。

5. Ace Project

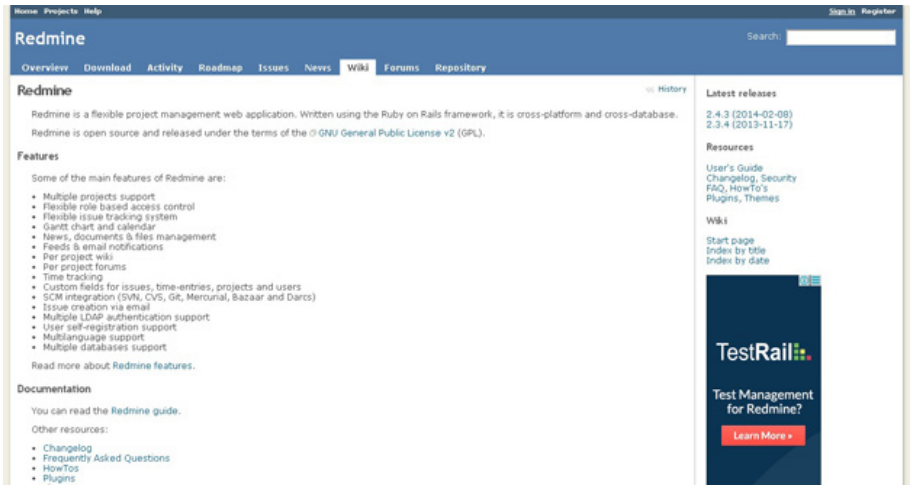
AceProject 提供了一系列最好的专业项目管理应用所具有的特性，例如 Microsoft Project，用一个更简单易用的任务管理系统。无论你是否是一个经验丰富的项目管理者，管理的团队或大或小，通过基于 web 的项目管理和协作系统会使你的项目变得更加有效率和生产力。简单的从电子表格转换到在线协作然后使用基于云的协作系统来获得无尽的好处吧。

6. PlanetSoho



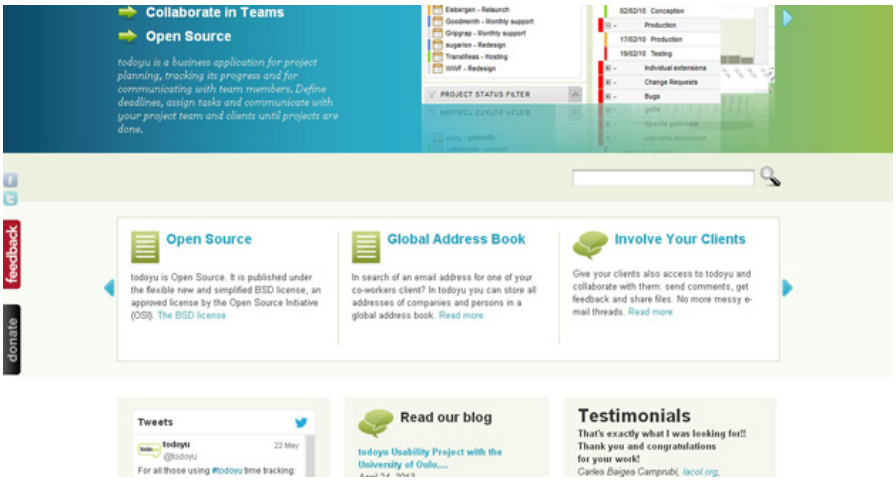
PlanetSohoOS 是一款集多功能于一体的在线业务管理程序。免费使用，并且集成了诸如项目、文档、客户及合同管理、沟通交流、发票等工具。一旦注册完成，欢迎你的是友好的仪表盘，这里显示了最新活动，快速链接，说明以及全部活动（此页可很容易的成为很多用户的开始页）。

7. Redmine



Redmine 是一款强大又免费的在线项目管理应用，可以作为其他收费应用的优良替代品。它几乎提供了你所需的一切功能，尽管这些功能不像那些昂贵的竞争对手那样华丽。通过 Redmine 可以同时管理大量的项目，定义个人角色以及随时分配任务到特定团队成员，还可以跟踪时间和问题，创建甘特图和日程表，创建项目 wiki 及论坛，以及管理文档和文件。

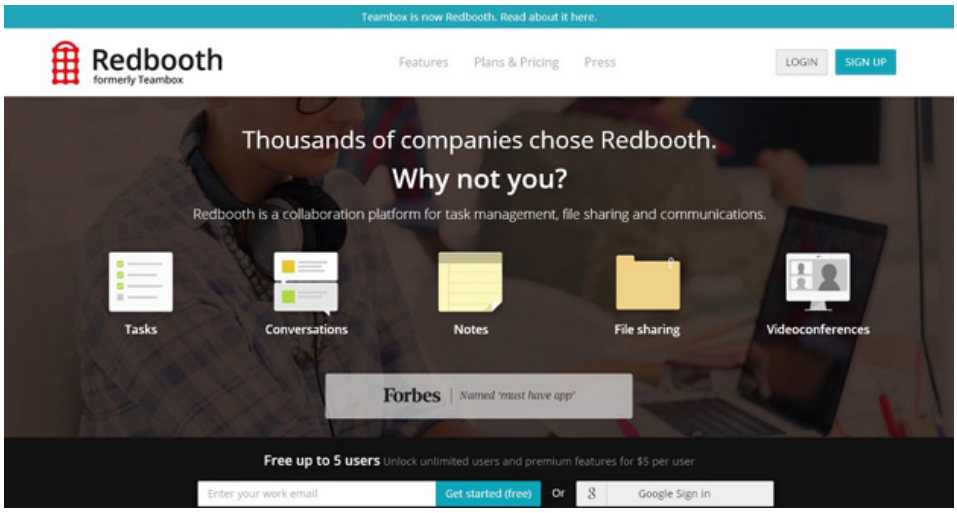
8 Todoyu



todoyu 是一个开源的任务/项目管理，时间跟踪和收集的应用系统，它是用 PHP 实现的。这个应用系统有一个现代的 Ajax 风格的接口，使用者和客户都可以有项目的角色。使用 todoyu,项目将可以分解成多个里程碑，每个里程碑都有可以分配到使用者的子任务。每个任务的时间都将记录下来用于将来的分析和财务核算。todoyu 可以在项目完成后自动生成发票清单并导出为 PDF 格式。

9 Redbooth

Redbooth 是一个开源的平台，有类似 Twitter 风格的接口，项目团队可以用它合



作项目。这个应用系统采用基于 Rails 的 Ruby 实现，除了开源版本之外还有托管版本。使用者可以创建无限数量的项目和任务，并把它们分配给同事，可以共享消息、文件等。使用者可以开启即时消息，可以创建页面，这些页面用于存储信息和互动，可以把客户包括到系统中从而方便的与他们合作。

10. 5pm

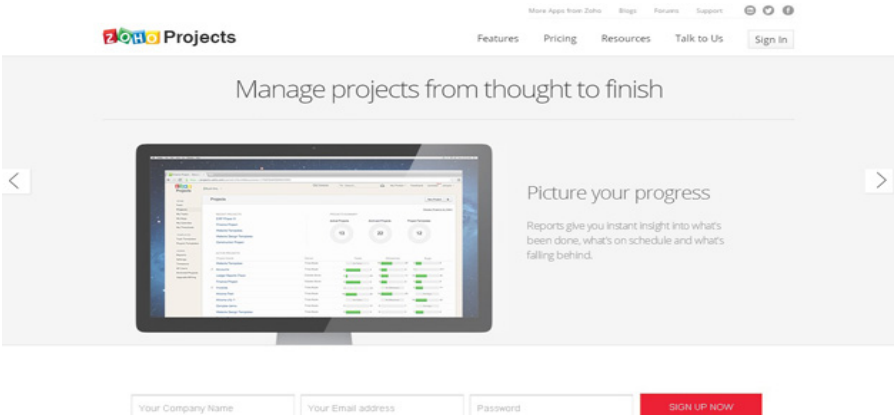
5pm 是一个直观的基于 Web 的项目管理工具。它灵活易用。在这里你可以通过邮件的方式请求执行特定的计划。也可以在夜间备份数据。或者进行 256 位的高等级数据安全加密。也提供了桌面跟踪工具。按时间线过滤用户，或者更多的特性在多种语言下都可用。

11. qdPM

qdPM 是一个基于 Web 的开源项目管理工具，适合于开发多个项目的小团队。具有充分的可配置性。你可以轻松的管理项目、任务、人员。通过传票系统与客户间的交互也集成到了任务管理中。

12. Zoho Projects

这是一个展示相关项目文档，已存储的



内容，最新提交的更新，大家交换意见以及完成工作的集中区域。选择 ZOHO 的最大理由是--超越计划，拥有完成项目的更多关键因素，计划，智慧工作，报告，绝对物有所值。

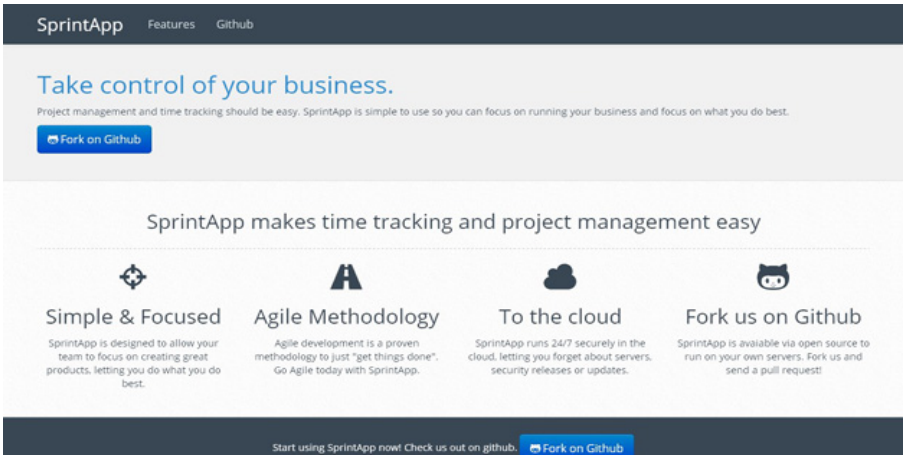
13. Springloops

这是一款为 web 开发者提供的源代码管理工具。你能并行处理代码以及安全的共享代码。此工具的主要特性有：强大的工作流，易用的管理，独一无二的代码浏览器，以及 Basecamp 集成。

14. Basecamp

Basecamp 集项目管理、协作和任务管理于一身，基于 web 提供服务，现已被数百万用户所信任。它提供了大量特性，包括共享文件，按时完成(Meet deadlines)，分配任务，集中反馈等等。Basecamp 以一种完全不同的视角诠释项目管理：关注沟通与协作。

15. Sprintapp



SprintApp 是一项专业的项目管理服务，同时还像开源软件一样共享了他的全部代码。 此应用由 Ruby on Rails 编写，界面现代，特性多样。工单/问题（Tickets/issues）是这个系统的核心。一组工单（tickets）构成了众多里程碑，这些里程碑被绑定到项目上。

原文链接：http://www.linuxeden.com/html/news/20140221/148629.html

Java 8 新闻：发布候选版面世、新的原子数、放弃简易实现

Java 8 的第一个发布候选版（RC）已于 2 月初面世。第一个发布候选版 b128 是 2 月 4 日发布的，第二个版本则于一周后在 OpenJDK 邮件列表中宣告问世。

Java 8 RC2 修复了新的 Comparator API 中的一个严重缺陷——新的 thenComparing()方法有一个不必要的类型约束。bug 报告指出：

在 java.util.Comparator 中，下面的方法要求类型 U 扩展 java.lang.Comparable。

<U extends Comparable<? super U>> Comparator<T> thenComparing(

Function<? super T, ? extends U> keyExtractor,

Comparator<? super U> keyComparator);

但是这一约束是不必要的，因为 keyComparator 用于比较的是提取出的 key 对象。

Java 8 RC2 还修复了在 Mac OS X 上的一个读权限问题。发布候选版可以从 https://jdk8.java.net/download.html 下载。

根据 JDK 8 的 bug 跟踪系统上的信息，Java 8 将于 3 月 17 日圣帕特里克节这天发布。截至本文写作时，还有 3 个问题尚未解决，都与文档有关。

在其他与 Java 8 有关的新闻中，Drew Stephens 最近发布的数据表明，Java 8 的原子数实现快了很多。此外，出于法律方面的原因，Mark Reinhold 提议放弃简易实现（Stripped Implementations）。

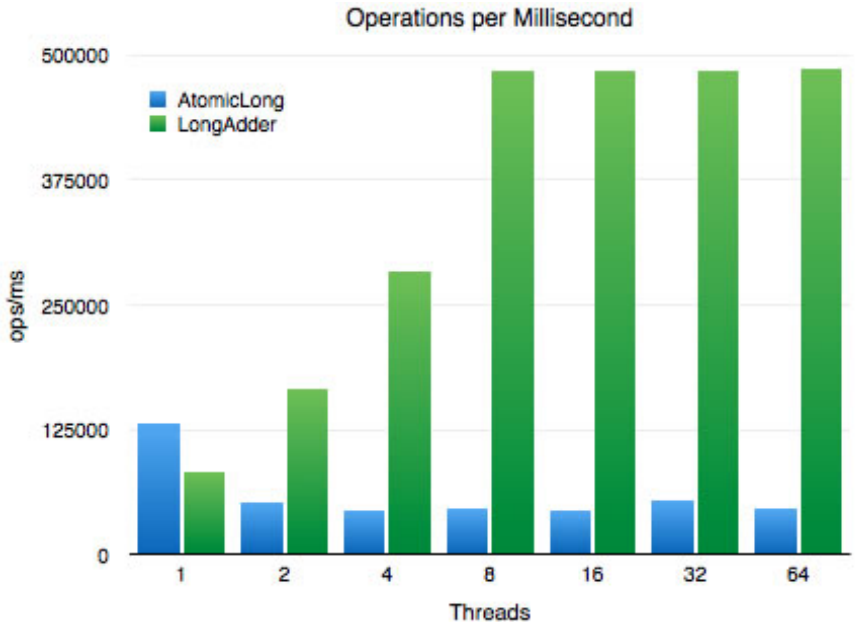
新的原子数实现

除了 Lambdas（335）和新的日期与时间 API（JSR 310），Java 8 还包含了对特定的多线程应用类非常重要的原子数实现。Palamino Labs 的负责人 Drew Stephens 最近撰文谈到了 LongAdder 和 DoubleAdder 的引入。

尽管 LongAdder 和 DoubleAdder 的引入不是那么光彩夺目，但是对于特定的多线程应用类而言，非常重要。这两个原子数实现在多线程竞态条件下提供了比 AtomicInteger 和 AtomicLong 更好的性能。

一些简单的基准测试就能说明其性能差别，下面的基准测试，我们使用了一

个 m3.2xlarge EC2 实例，它可以访问一个 Intel Xeon E5-2670 处理器的全部 8 个核心。



在单线程条件下，新的 LongAdder 会慢 1/3，但是当多个线程竞争着增加字段时，LongAdder 就体现出了其价值。请注意，每个线程唯一要做的是尝试增加计数器——这是一个最极端的综合基准测试。这里的竞争比你现实中可能看到的大部分应用更为强烈，但有时你确实需要这类共享计数器，而 LongAdder 能提供很大的帮助。

Drew 继续演示了 AtomicLong，它在单线程条件下快一点。然而，在使用两个线程时，它要慢 4 倍；当线程数与核数相同时，它几乎慢 5 倍。他还指出，“当线程数超过 CPU 的物理核数时，LongAdder 的性能就保持恒定了”。

放弃简易实现

简易实现是 Java 8 的一个计划特性，支持将 Java SE 的定制实现与要基于它运行的应用打包到一起。不依赖应用所使用代码的元素可以移除。对于想把 Java 嵌入到设备中的应用，这类实现可能是有用的（比如家用设备）。

Mark Reinhold 最近提议从 Java SE 8 中去掉简易实现。他援引了法律问题作为理由。

为保持兼容性并防止分裂，Java SE 8 的简易实现特性需要对 TCK 许可做一些重大的修改。

我已经和 Oracle 的法务部门就这些修订共同工作过一段时间。我们有一个初

始草案，但是现在遗憾的是，我认为专家组成员、JCP 执行委员会成员和其他各方已经没有足够的时间审阅和评注这些修改了。

因此我建议从 Java SE 8 中去掉简易实现特性。这只需要修改规范和 TCK 规则，不需要修改参考实现或实际的 TCK 测试。

Reinhold 还写道，他认为简易实现对于 Java 平台的未来很是重要，该特性可能会在 Java SE 9 之前的版本中加入。

Java 8 的发布已经近在咫尺。更易用的日期、闭包、更好的并发和一个新的 JavaScript 引擎，离我们只有一个月之遥了！你会升级吗？如果不升级，是有技术方面的原因阻止你升级吗？

原文链接：http://www.infoq.com/cn/news/2014/02/java8-release-candidates

2013 年 StackOverflow 开发者调查：JS 最火

截止 2013 年，Stack Overflow 社区的月访问量从 2150 万次增长到了 2690 万次，访问者分别来自全球的 242 个国家。为了维持社区的增长，我们做了很多的努力——Careers 2.0 已经有了法语和德语两个本地化版本，我们正致力于为整个 Stack Overflow 网络开发 iOS 和 Android 应用，并且 Stack Overflow 历史上的首次本地化尝试——葡萄牙语版 Stack Overflow——已经处于 Beta 状态。为了让我们更好地为 Stack Overflow 社区和用户服务，我们每年都会进行一项调查以了解用户心中的期望，用户对网站的使用反馈以及用户心中的其他想法。

2013 年，我们分析了 96 个国家的 7,500 份调查样本。作为对你们参与调查的酬谢，我们已经向 Stack Exchange Charities 捐款 12,000 美元。

主要调查结果

1. 这是我们呼唤移动端的第二年，而它仍在快速增长。

尽管受调查者中只有 7.9%的人将自己归为移动应用开发者，但是大多数的受调查者（51.5%!）表示他们的公司开发了原生移动应用。与 2012 年相比这是一个增长点，那时只有 48.2%的受访者表示他们的公司开发了原生移动应用。

2. Android 势头持续上升，而 iPhone 则有所下降

Android 手机不仅成为最受欢迎的移动设备，63.8%的受调查者表示他们拥有一部 Android 手机，而且成为了最受欢迎的本地移动平台，30.7%的受调查者支持 Android 手机应用。与 2012 年相比，iPhone 则失去了更多的开发者，2013 年只有 30.7%的受调查者表示他们拥有一部 iPhone 手机，而 2012 年时该比例为 35.2%。

3. 远程办公

随着 Stack Exchange 团队的快速增长，我们有越来越多的员工开始远程办公，因此我们特地在调查中加入了远程办公相关的问题。然而只有 10.6%的受调查者表示他们是全职远程办公，不过 63.9%的被调查者表示他们至少会偶尔远程办公。

下面是一份特制的信息图，图中总结了我们这次调查的主要结果。如果你想亲自对结果数据做分析，你可以在这里下载调查结果数据。

以下由伯乐在线摘录信息图中的部分数据：

编程经验年限：

少于 2 年：14.8%

2 – 5 年：32.7%

6 – 10 年：24.7%

11 年以上：27.7%

2013 年热门语言/技术：

JavaScript

SQL

Java

C#

PHP

Python

C++

C

Objective-C

Node.js

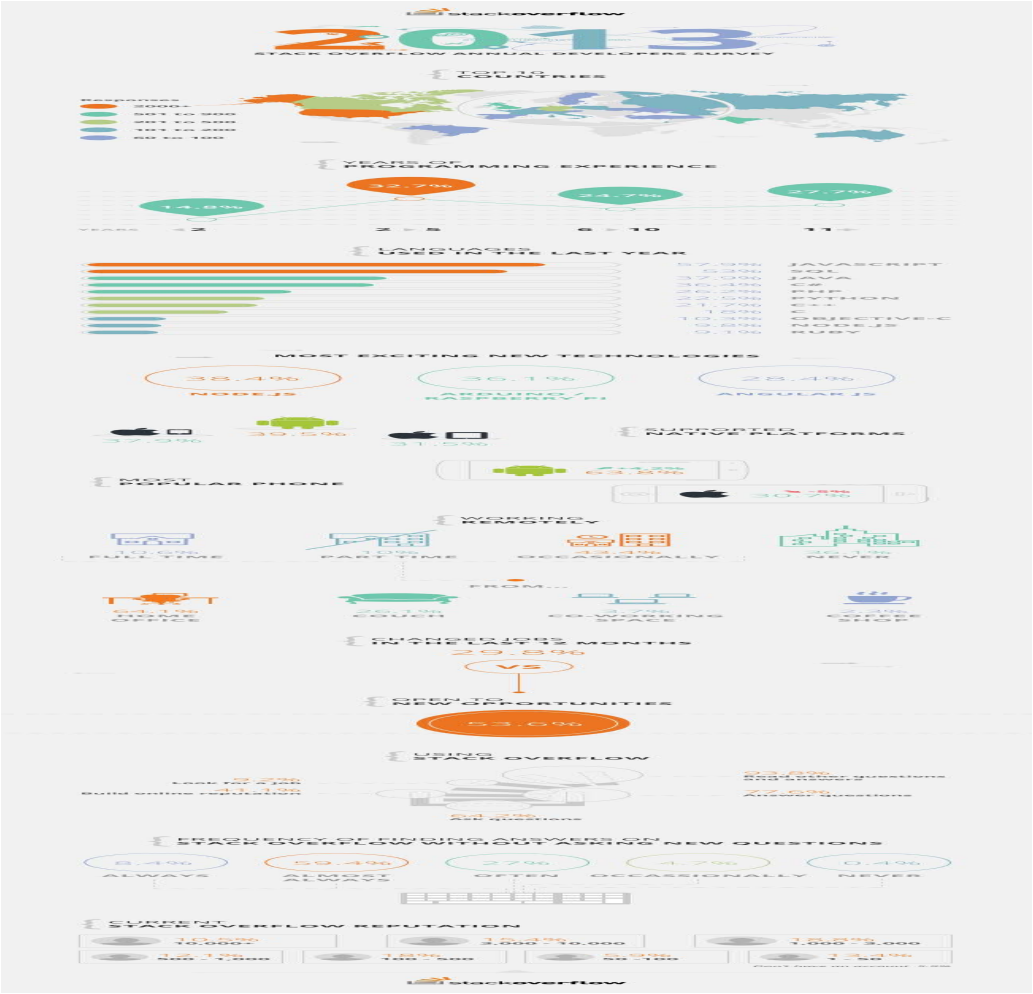
Ruby

最令人兴奋的新技术：

Node.JS：38.4%

Arduino / Rassberry Pi ：36.1%

AngularJS：28.4%



（图放不下就缩放了，可以下载放大！）

原文链接:http://www.iteye.com/news/28781-stack-overflow-user-survey-results

[前端开发]

再谈榔头和钉子

不久前写过一篇《给我一把榔头，满世界都是钉子》，从算法和数据结构的角度谈了谈对于问题和问题解决的工具这两方面我的看法；而最近看到了这样的代码，一个表格，单数行和双数行的样式不同，于是有程序员这样写道：

```
1    vartrs = $("#spreadSheet tr");
2    for(vari=0; i<trs.size(); i++){
3        if(i%2)
4            $(trs.get(i)).children("td").css("color","RED");
5        else
6            $(trs.get(i)).children("td").css("color","GREEN");
7    }
```

从功能实现上看，这是一点都没有问题的，JQuery 这把“榔头”，确实是把这个“钉子”给砸进墙里面去了。但问题在于，这是个图书钉，使用了一个建筑工地上用的超大号榔锤。样式的问题，当然优先考虑 CSS 去解决：

```
1    #spreadSheet tr:nth-child(odd) td {
2        color : GREEN;
3    }
4    #spreadSheet tr:nth-child(even) td {
5        color : RED;
6    }
```

这其实是一个很常见的问题，不同的榔头，应该用来解决不同的问题。类似这样的问题有很多，再比如下面这几组列表的混用：

ul/li：无序列表

ol/li：有序列表

dl/dt/dd：普通列表

w3cshool 中文网站上面有 HTML 标签的详尽列表，虽说基础，但是全部掌握这些语义并在合适的时机使用也并不容易。

如果说，榔头选得不好，依然把钉子砸进去了，只是这个过程有些别扭，那已经是很不错的一个结果了。更糟的可能是，砸完钉子以后无尽的后遗症。我记得刚工作的时候，第一个项目是中国移动的彩铃项目，这类项目都是面向电信运营商的，业务逻辑可谓相当复杂，整个系统大概超过了六十万行代码，但是业务逻辑居然是写在存储过程里的。开山鼻祖的程序员第一次拿存储过程这个榔头砸问题的时候，他大概也没有想到，这会成为名副其实的“maintenance burden”，后续无尽痛苦的源泉。

关于编程范型

接着我想谈一谈设计模式和编程范型。抽象地说，它们是两种不同角度的对榔头的分类方式。更多的人非常熟悉设计模式（Design Pattern）的含义，大多数设计模式和语言类型、语言本身无关，掌握得好的程序员可以写出简洁、解耦、易维护的代码；半吊子程序员也可以写出概念堆叠、过度设计的代码。

但是编程范型（Programming Paradigm）则往往和语言本身的特性强相关，一种特定的语言，只适用于一种或几种编程范型。简单地说，它类似于一种编程风格，换一种说法，它是问题解决方案落到代码上的表现形式。但是，能否恰当地使用编程范型，决定了能否写出清晰、高效的代码。

我曾经在《编程的未来》里面提到过编程范型的进化：

很多时候程序员会觉得，算法还是不容易转变成代码，即便是简单的算法，思路简单的纸上实现，变成代码却比较冗长。我觉得大部分情况下这不是你编码技巧的问题，而是编程语言的问题——换句话说，如果你使用一种合适范型的编程语言，兴许就可以轻松解决这个问题——即便这样的语言并不一定好找，并不一定容易设计。

也使用了 Prolog 作为例子。在维基百科的链接上，可以找得到很多编程范型的归类，最常见的几个说出来也会觉得耳熟能详：

声明式编程

事件驱动编程

面向切面编程

管道编程

……

学习一种新的语言，其中一项重要的意义也在于此；有的框架，特别是提供了 DSL 特性的框架，也具备这样的意义。最典型的例子就是写一写前端代码对于程序员来说的意义，我写过一篇《程序员，都去写一写前端代码吧》，但是其中漏掉了一点，前端的代码（HTML+CSS+JavaScript）带来的编程范型是非常丰富的，尤其是 JavaScript，你可以对比一下 JavaScript 的开源库和 Java 的开源库，Java 的开源库更多的是注重与功能和框架设计，而 JavaScript 的开源库则提供了大量崭新的写代码的风格。我可以不做前端的工作，但是我依然会学 JavaScript。

举例来说，D3，我以前的一位经理说，D3 实在是太反直觉了，不适合用到我们的项目里面去。说反直觉那确实也是正确的，但是很多情况下这是建立在人已有认识的基础上的，一旦熟悉并习惯了 D3 的编程范型（接近于声明式，核心是几个不同的状态，加上状态之间的变迁，而这些变迁的过程可以绑定上丰富的行为），你会发现它的代码可以写得如此优雅和简洁。

原文链接：http://www.raychase.net/2261

短小强悍的 JavaScript 异步调用库

对于博文 20 行完成一个 JavaScript 模板引擎 的备受好评我感到很惊讶,并决定用此文章介绍使用我经常使用的另一个小巧实用的工具.我们知道,在浏览器中的 JavaScript 绝大部分的操作都是异步的(asynchronous),所以我们一直都需要使用回调方法,而有时不免陷入回调的泥淖而欲死欲仙.

假设我们有两个 functions ,我们顺序地在一个后面执行完后调用另一个。他们都操作同一个变量。第一个设置它的值,第二个使用它的值。

```
[javascript] view plaincopy
var value;
var A = function() {
    setTimeout(function() {
        value = 10;
    }, 200);
}
var B = function() {
    console.log(value);
}
```

那么,现在如果我们运行 A();B(); 我们将在控制台看到输出为 undefined . 之所以会这样是因为 A 函数使用了异步方式设置 value 。我们能做的就是传一个回调函数给 A,并让函数 A 在执行完后执行回调函数。

```
[javascript] view plaincopy
var value;
var A = function(callback) {
   setTimeout(function() {
      value = 10;
      callback();
   }, 200);
};
```

```
var B = function() {
    console.log(value);
};
A(function() {
    B();
});
```

这样确实有用,但想象一下加入我们需要运行 5 个或更多方法时将会发生什么。一直传递回调函数将会导致混乱和非常不雅观的代码。

好的解决办法是写一个工具函数,接受我们的程序并控制整个过程。让我们先从最简单的开始:

[javascript] view plaincopy

```
var queue = function(funcs) {
        //  接下来请看,董卿???
}
```

接着，我们要做的是通过传递 A 和 B 来运行该函数 - queue([A, B])。我们需要取得第一个函数并执行它。

[javascript] view plaincopy

```
var queue = function(funcs) {
        var f = funcs.shift();
        f();
}
```

如果执行这段代码,您将看到一个 TypeError: undefined is not a function。这是因为 A 函数没收到回调参数但却试图运行它。让我们换一种调用方法。

[javascript] view plaincopy

```
var queue = function(funcs) {
        var next = function() {
                // ...
        };
        var f = funcs.shift();
```

```
        f(next);
    };
```

在 A 执行完后会调用 next 方法。将下一步操作放在 next 函数列表中是个很好的做法。我们可以将代码归拢在一起,而且我们能够传递整个数组(即便数组中有很多函数等待执行)。

[javascript] view plaincopy

```
var queue = function(funcs) {
        var next = function() {
                var f = funcs.shift();
                f(next);
        };
        next();
};
```

到了这一步,我们基本上达到了我们的目标。即函数 A 执行后,接着会调用 B,打印出变量的正确值。这里的关键是 shift 方法的使用。它删除数组的第一个元素并返回该元素。一步一步执行下去, funcs 数组就会变成 empty(空的)。所以,这可能会导致另一个错误。为了证明这一观点,让我们假设我们仍然需要运行这两个功能,但我们不知道他们的顺序。在这种情况下,两个函数都应该接受回调参数(callback)并执行它。

[javascript] view plaincopy

```
var A = function(callback) {
        setTimeout(function() {
                value = 10;
                callback();
        }, 200);
};
var B = function(callback) {
        console.log(value);
        callback();
```

```
};
```

当然，我们会得到 TypeError: undefined is not a function.

要阻止这一点,我们应该检查 funcs 数组是否为空。

```
[javascript] view plaincopy
var queue = function(funcs) {
    (function next() {
        if(funcs.length > 0) {
            var f = funcs.shift();
            f(next);
        }
    })();
};
```

我们所做的就是定义 next 函数并调用它。这种写法减少了一点代码。

让我们试着想象尽可能多的情况。比如当前执行功能的 scope 。函数内的 this 关键字可能指向了全球的 window 对象。,如果我们可以设置自己的 scope 当然是件很酷的事情。

```
[javascript] view plaincopy
var queue = function(funcs, scope) {
    (function next() {
        if(funcs.length > 0) {
            var f = funcs.shift();
            f.apply(scope, [next]);
        }
    })();
};
```

我们为这个 tiny 类库增加了一个参数。接着我们使用 apply 函数,而不是直接调用 f(next),来设置 scope 并将参数 next 传递进去。同样的功能,但漂亮多了。

我们需要的最后一个特性,就是是函数间传递参数的能力。当然我们不知道具体

会有多少参数将被使用。这就是为什么我们需要使用 arguments 变量的原因。你可能知道,该变量在每个 JavaScript 函数中都是可用的,代表了传进来的参数。它就和一个数组差不多,但不完全是。因为在 apply 方法中我们需要使用真正的数组,使用一个小窍门来进行转换。

[javascript] view plaincopy

```
var queue = function(funcs, scope) {
    (function next() {
        if(funcs.length > 0) {
            var f = funcs.shift();
            f.apply(scope,
[next].concat(Array.prototype.slice.call(arguments, 0)));
        }
    })();
};
```

下面是测试的代码:

[javascript] view plaincopy

```
// 测试代码
var obj = {
    value: null
};

queue([
    function(callback) {
        var self = this;
        setTimeout(function() {
            self.value = 10;
            callback(20);
        }, 200);
```

```
        },
        function(callback, add) {
            console.log(this.value + add);
            callback();
        },
        function() {
            console.log(obj.value);
        }
    ], obj);
```

执行后的输出为:

[plain] view plaincopy

```
30
10
```

为了代码的可读性和美观，我们将部分相关的代码移到一行内:

[javascript] view plaincopy

```
var queue = function(funcs, scope) {
    (function next() {
        if(funcs.length > 0) {
            funcs.shift().apply(scope || {},
[next].concat(Array.prototype.slice.call(arguments, 0)));
        }
    })();
};
```

你可以 点击这里查看并调试相关代码 ,完整的测试代码如下:

[javascript] view plaincopy

```
var queue = function(funcs, scope) {
    (function next() {
        if(funcs.length > 0) {
```

```
                funcs.shift().apply(scope || {},
[next].concat(Array.prototype.slice.call(arguments, 0)));
            }
        })();
    };
    var obj = {
        value: null
    };
    queue([
        function(callback) {
            var self = this;
            setTimeout(function() {
                self.value = 10;
                callback(20);
            }, 200);
        },
        function(callback, add) {
            console.log(this.value + add);
            callback();
        },
        function() {
            console.log(obj.value);
        }
    ], obj);
```

原文链接：http://blog.csdn.net/renfufei/article/details/19428719

BigPipe 学习研究

1. 技术背景 FaceBook 页面加载技术

试想这样一个场景，一个经常访问的网站，每次打开它的页面都要要花费 6 秒；同时另外一个网站提供了相似的服务，但响应时间只需 3 秒，那么你会如何选择呢？数据表明，如果用户打开一个网站，等待 3~4 秒还没有任何反应，他们会变得急躁，焦虑，抱怨，甚至关闭网页并且不再访问，这是非常糟糕的情况。所以，网页加载的速度十分重要,尤其对于拥有遍布全球的 5 亿用户的 Facebook(全球最大的社交服务网站)这样的大型网站，有着大量并发请求、海量数据等客观情况，速度就成了必须攻克的难题之一。

2010 年初的时候，Facebook 的前端性能研究小组开始了他们的优化项目，经过了六个月的努力，成功的将个人空间主页面加载耗时由原来的 5 秒减少为现在的 2.5 秒。这是一个非常了不起的成就，也给用户来带来了很好的体验。在优化项目中，工程师提出了一种新的页面加载技术，称之为 Bigpipe。目前淘宝和 Facebook 面临的问题非常相似：海量数据和页面过大，如果可以在详情页、列表页中使用 bigpipe，或者在 webx 中集成 bigpipe，将会带来明显的页面加载速度提升。

2. 相关介绍

2.1 网站前端优化的重要性

《高性能网站建设指南》一书中指出，只有 10%~20%的最终用户响应时间是花费在从 Web 服务器获取 HTML 文档并传送到浏览器中的。如果希望能够有效地减少页面的响应时间，就必须关注剩余的 80%~90%的最终用户体验。做个比较，如果对后台业务逻辑进行优化，效率提高了 50%，但最终的页面响应时间只减少了 5%~10%，因为它所占的比重较少。如果对前端进行性能优化，效率提升 50%，则会使最终页面响应时间减少 40%~45%。这是多么可观的数字！另外，前端的性能优化一般比业务逻辑的优化更加容易。所以，前端优化投入小，见效快，性价比极高，需要投入更多的关注

2.2 BigPipe 与 AJAX

Web2.0 的重要特征是网页显示大量动态内容，即 web2.0 注重网页与用户的

交互。其核心技术是 AJAX，如今所有主流网站都或多或少使用 AJAX。与 AJAX 类似，BigPipe 实现了分块儿的概念，使页面能够分步输出，即每次输出一部分网页内容。接下来讨论 BigPipe 与 AJAX 的区别。

简单的说，BigPipe 比 AJAX 有三个好处：

1. AJAX 的核心是 XMLHttpRequest，客户端需要异步的向服务器端发送请求，然后将传送过来的内容动态添加到网页上。如此实现存在一些缺陷，即发送往返请求需要耗费时间，而 BigPipe 技术使浏览器并不需要发送 XMLHttpRequest 请求，这样就节省时间损耗。

2. 使用 AJAX 时，浏览器和服务器的工作顺序执行。服务器必须等待浏览器的请求，这样就会造成服务器的空闲。浏览器工作时，服务器在等待，而服务器工作时，浏览器在等待，这也是一种性能的浪费。使用 BigPipe，浏览器和服务器可以并行同时工作，服务器不需要等待浏览器的请求，而是一直处于加载页面内容的工作阶段，这就会使效率得到更大的提高。

3. 减少浏览器发送到请求。对一个 5 亿用户的网站来说，减少了使用 AJAX 额外带来的请求，会减少服务器的负载，同样会带来很大的性能提升。

基于以上三点，Facebook 在进行页面优化时采用了 BigPipe 技术。目前淘宝主搜索结果页中，需要加载类目，相关搜索，宝贝列表，广告等内容，前端这里使用 php 的 curl 的批处理来并发的访问引擎获取相应的数据，并进行分步输出。这种模式还是与 bigpipe 有些不同，这点后面会讲到。一般来讲，在页面比较大，而且比较复杂，样式表和脚本比较多的情况下，使用 BigPipe 来优化输出页面是比较合适的。另外非常重要的一点，BigPipe 并不改变浏览器的结构与网络协议，仅使用 JS 就可以实现，用户不需要做任何的设置，就会看到明显的访问时间缩短。

3 目前的问题

接下来讨论现有的瓶颈。面对网页越来越大的情况，尤其是大量的 css 文件和 js 文件需要加载，传统的页面加载模型很难满足这样的需求，直接结果就是页面加载速度变慢，这绝不是我们希望看到的。目前的技术实现中，用户提出页面访问请求后，页面的完整加载流程如下：

1. 用户访问网页，浏览器发送一个 HTTP 请求到网络服务器

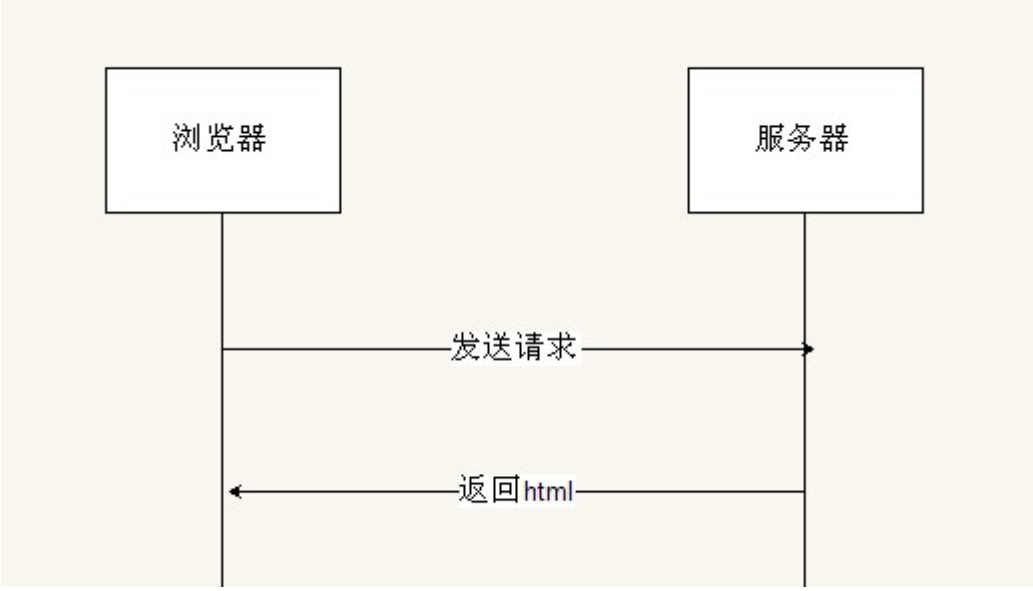
2. 服务器解析这个请求，然后从存储层去数据，接着生成一个 html 文件内容，并在一个 HTTP Response 中把它传送给客户端

3. HTTP response 在网络中传输

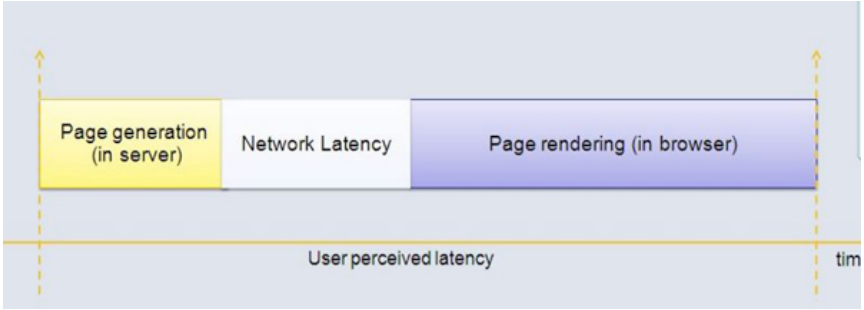
4. 浏览器解析这个 Response ，创建一个 DOM 树，然后下载所需的 CSS 和 JS 文件

5. 下载完 CSS 文件后，浏览器解析他们并且应用在相应的内容上

6. 下载完 JS 后，浏览器解析和执行他们

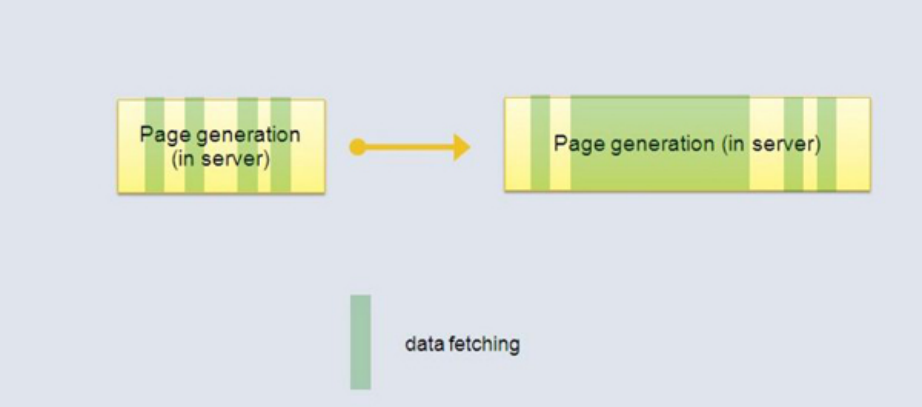


完整流程见图 1.图中左侧表示服务器，右侧表示浏览器。浏览器先发送请求，然后服务器进行查找数据，生成页面，返回 html 代码，最后浏览器进行渲染页面。这种模式有非常明显的缺陷：流程中的操作有着严格的顺序，如果前面的一个操作没有执行结束，后面的操作就不能执行，即操作之间是不能重叠。这样就造成性能的瓶颈：服务器生成一个页面的内容时，浏览器是空闲的，显示空白内容；而当浏览器加载渲染页面内容时，服务器又是空闲的，时间与性能的浪费由此产生。



考虑图 2 中现有的服务模型，横轴表示花费的时间。黄色表示在服务器的生成页面内容的时间，白色表示网络传输时间，蓝色表示在浏览器渲染页面的时间。可以看出，现有的模式造成很大的时间浪费。 考虑图 3 中的情况，图中绿色表示服务器从春储层取查数据花费的时间，在海量数据下，当执行一条很费时的查询语句时（如下图右侧），服务器就就阻塞在那 里没有其他操作，而浏览器更是得不到任何反馈。这会造成非常不友好的用户体验，用户不知道什么原因使他们等待很长时间。

图 3.



4 BigPipe 思想与原理

面对上述问题，我们看下 BigPipe 的解决办法。BigPipe 提出分块的概念，即根据页面内容位置的不同，将整个页面分成不同的块儿– 称为 pagelet。该技术的设计者 Changhao Jiang 是研究电子电路的博士，可能从微机上得到了启发，将众多 pagelet 加载的不同阶段像流水线一样在浏览器和服务器上执行，这样就做到了浏览器和服务器的并行化，从而达到重叠服务器端运行时间和浏览器端运行时间的目的。使用 BigPipe 不仅可以节省时间，使加载的时间缩短，而且可以同过 pagelet 的分步输出，使一部分的页面内容更快的输出，从而获得更好的用户体验。BigPipe 中，用户提出页面访问请求后，页面的完整加载流程如下:

1. Request parsing：服务器解析和检查 http request
2. Datafetching：服务器从存储层获取数据
3. Markup generation：服务器生成 html 标记
4. Network transport ： 网络传输 response
5. CSS downloading：浏览器下载 CSS
6. DOM tree construction and CSS styling:浏览器生成 DOM 树，并且使用 CSS

7. JavaScript downloading: 浏览器下载页面引用的 JS 文件

8. JavaScript execution: 浏览器执行页面 JS 代码

这个 8 个流程几乎与上文中提到现有的模式没有区别，但这整个流程只是一个 pagelet 的完整流程，而多个 pagelet 的不同操作阶段就可以像流水线一样进行执行了。

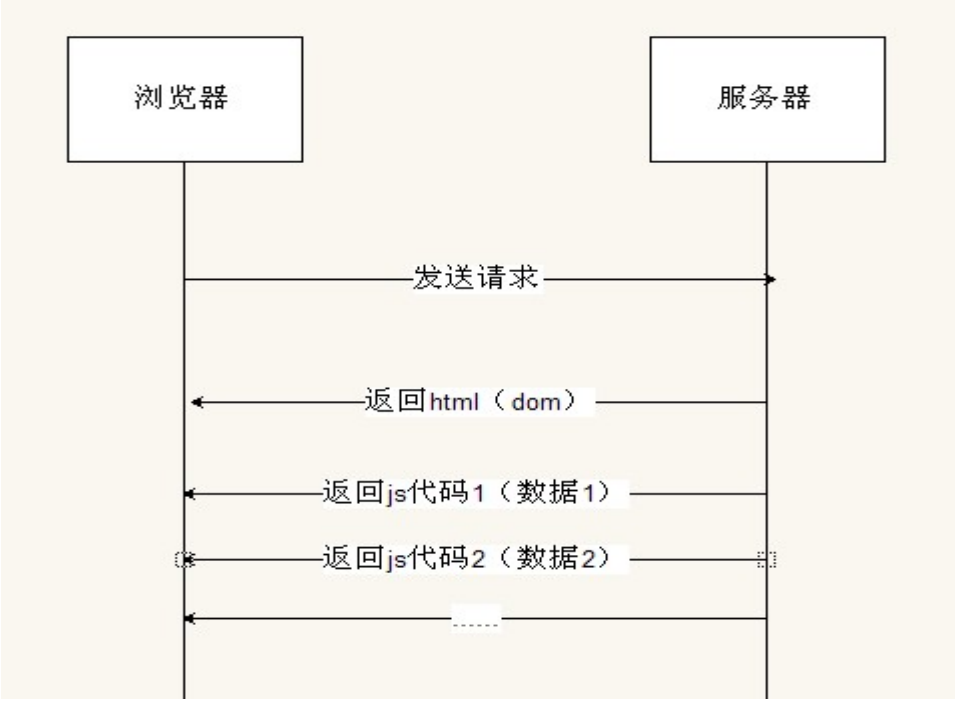
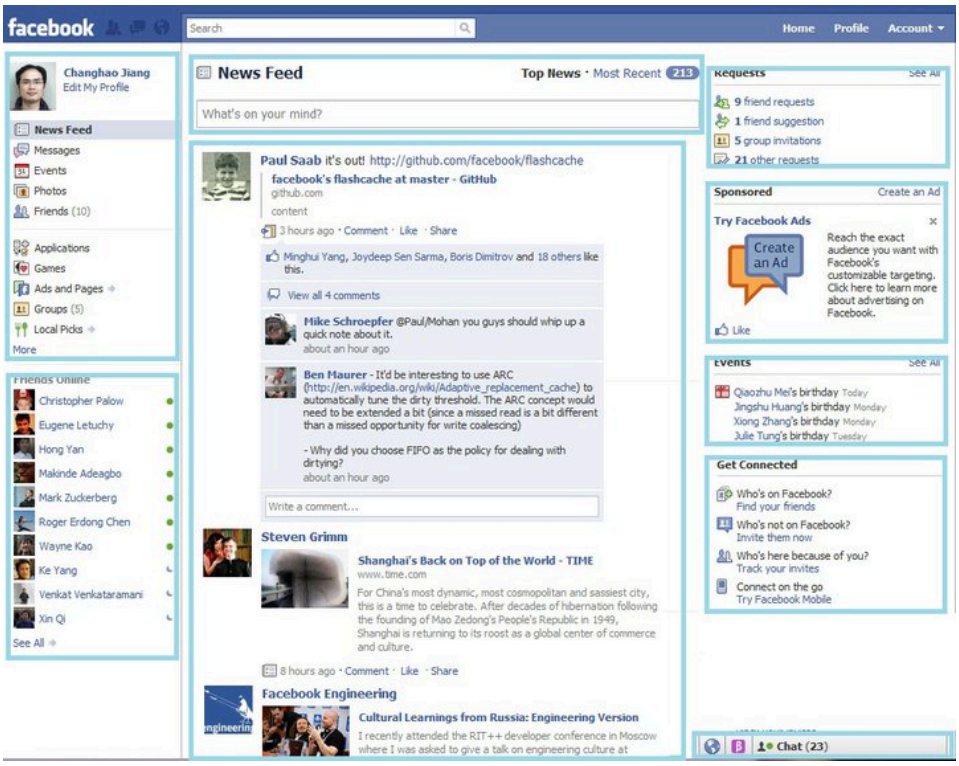


图 4 中，可以看出 BigPipe 对原有的模式进行的改进。浏览器发送访问请求，然后浏览器分步返回不同的 pagelet 的内容，具体实现将在后面介绍.考虑图 5 中的改进，BigPipe 打破了原有的顺序执行，将页面分成不同的 pagelet ，如此一来，所有的 pagelet 的执行时间累加起来还是原有的时间。但是， 通过叠加不同 pagelet 的不同阶段的执行时间，使总的运行时间大大减少，这就是 Bigpipe 减少页面加载时间的秘密。

FaceBook 的页面被分成了很多不同的 pagelets，如图：

图 5



5 BigPipe 实现原理

了解了 BigPipe 的核心思想后，我们讨论它的实现原理。当浏览器访问服务器时，服务器接受请求并对其进行检查。如果请求有效，服务器端不做任何的查询，而是立刻返回一个 http request 给浏览器，内容是一段 html 代码，包括 html<head> 标签和<body> 标签的一部分。<head>标签包括 BigPipe 的 js 文件和 css 文件，这个 js 文件用来解析后面接收的 http response，因为后面传输的内容都为 js 脚本。未封闭的<body>标签中，是显示页面的逻辑结构和 pagelet 的占位符的模板，例如：

```
<body>
<div></div>
<div></div>
<div></div>
<div>
<div>
<div id=”hotnews”></div>
```

上述模板使用 css-div 描述了页面的结构，不同的 div 标签对应不同的 pagelet，id 对应了 pagelet 的名称。将这个 response 返回给浏览器后，服务器开始对每个 pagelet 的内容进行查询，加载，生成。当一个 pagelet 的内容生成好，立刻调用 flush()函数，将其返回给客户端，传输的数据是以 json 格式的，包括这个 pagelet 需要的 CSS 和 JS，以及 html 内容和一些元数据。例如：

<script type=”text/javascript”>

big_pipe.onPageletArrive(

{id:”pagelet_composer”,

content:”<HTML>”,

css:”[..]“,

js:”[..]“,

…}

);

</script>

其中”content”表示这个 pagelet 的内容，是 html 源码，特殊字符如“”/需要进行转义；”id”表示 content 要显示的位置，即为对应的 pagelet 的 id 标签；”css”表示需要下载的 CSS 资源的路径；”js”表示需要下载的 JS 脚本的路径。为了避免文件路径过长，所以在前面需要对 css 和 js 文件的路径进行转换，转换后为 5 位字符串：不同的 pagelet 可能会加载同一个 css 或 js 文件，所以要避免重复下载。

虽然每个 pagelet 都有要加载的 js 文件，但是所有的 js 文件都是在最后加

载，这样有利于加快页面加载速度。客户端，当通过调用“onPageletArrive（json）”函数，第一次影响传输的 JS 脚本中的函数解析了传入的 json 数据，接着下载需要的 CSS，然后把 html 内容显示到响应的 DIV 标签位置上。多个 pagelets 的 CSS 文件可以同时下载，CSS 下载完成的 pagelet 先显示。

在 BigPipe 中，js 被给予了比 CSS 和 content 更低的优先级。这样， 只有当所有的 pagelets 都显示了，BigPipe 才开始去下载 JS 文件。所有的 JS 文件都下载完成后，Pagelets 的 JS 初始化代码开始执行，按照下载完成时间的先后顺序。在这个高度并行的系统中，几个的 pagelet 所要执行的不同的阶段可以同时执行。例如，浏览器可以给两个 pagelets 下载 CSS 资源，同时浏览器可以渲染另外一个 pagelet 的内容，同时服务器仍然在为另一个 pagelet 生成 html 源码。从用户的角度看来，页面时逐步呈现的。初始的页面显示的更快，可以有效减短用户感觉到的延迟。

6 BigPipe 实现问题讨论

6.1 服务器端的并行化

理想情况下，服务器端的实现是并行处理不同的 pagelet 的内容，这样可以提升性能。服务器并发处理多个 pagelet 的内容时，一个 pagelet 内容生成好了，立刻将其 flush 给浏览器。但是 PHP 是不支持线程，所以服务器无法利用多线程的概念去并发的加载多个 pagelet 的内容。对于小型网站来说，使用串行的加载 pagelet 的内容就已经可以达到优化的要求了。对于大型网站，为了达到更快的速度，服务器端可以选择并发的独立不同的 pagelet 的内容，具体实现有以下几种方式：

1.java 多线程。后台逻辑使用 java，可以使用 java 的多线程机制去同时加载不同的 pagelet 的内容，加载完成后加页面内容返回给浏览器。在最后的引用部分可以看到网上用 java 多线程实现的例子。

2.使用 PHP 实现。PHP 不支持线程，无法像 java 使用多线程的机制来并发处理不同 pagelet 的内容。但是，Facebook 和淘宝主搜索的业务逻辑是用 PHP 实现的，所以我们必须考虑如何在 PHP 下完成并发处理。PHP 扩展中有 curl 模块，可以在该模块中 curl_multi_fetch()函数进行批处理请求，把本来应该串行的请求访问并发的执行。可以这样写：

```
1           do {
2
3           $mrc = curl_multi_exec($mh, $active);
4
5           }
6
7           while($mrc==CURLM_CALL_MULTI_PERFORM);
8
9           while ($active &amp;&amp; $mrc == CURLM_OK){
10
11          if (curl_multi_select($mh) != -1){
12
13          do {
14
15          $mrc = curl_multi_exec($mh,$active);
16
17          }
18
19          while($mrc==CURLM_CALL_MULTI_PERFORM);
20
21          }
22
23          }
```

但是会碰到一个问题，多个请求是同时返回结果的。当所有的 pagelet 的页面请求所消耗的时间差不多时，可以达到很好的性能，但是当有的消耗时间很长（执行一条复杂的查询）的情况下，批处理就会阻塞在那里，等每个请求都返回结果了才结束。而在这段时间致服务器会阻塞在那里不返回任何内容，而浏览器更是没有响应，这样就违背了 BigPipe 的原理。另外一种实现方法是使用 stream_select

函数。跟上一种方法类似，不过可以使用 PHP5 中新增的 stream_socket_client()函数链接而不是之前 PHP 函数中的 fsocketopen()函数。

```
1		while (count($sockets)) {
2
3		$read = $write = $sockets;
4
5		$n = stream_select($read,$write, $e, $timeout);
6
7		if ($n &gt; 0) {
8
9		foreach ($read as $r) {
10
11		$id = array_search($r, $sockets);
12
13		$data = fread($r, 8192);
14
15		if (strlen($data) == 0) {
16
17		fclose($r);
18
19		unset ($sockets[$id]);
20
21		}else {
22
23		$retdata[$id] .= $data;
24
25		}
26
27		}
```

```
28
29              $retdata[$id] = preg_replace('/^HTTP(.*?)\r\n\r\n/is',<em>,
30      $retdata[$id]);</em>
31
32              foreach ($write as $w) {
33
34              if (!is_resource($w))continue;
35
36              $id = array_search($w, $sockets);
37
38              fwrite($w, "GET /" . $url[$id] . "HTTP/1.0\r\nHost: " .
39      $hosts[$id] ."\r\n\r\n");
40
41              $status[$id] = 1;
42
43              }
44
45              }else {
46
47              break;
48
49              }


                }
```

这样实现也可以做到服务器的并发访问，但是会碰到和上一种方法同样的问题：服务器的阻塞问题。所以，可以采用另一种方法，用多进程模拟多线程。使用 PHP 的扩展模块 pctnl 模块中的 pcntl_fork（）函数来生成子进程， 用不同的子进程去处理不同的 pagelet 的页面内容。如果子进程返回内容，则返回给浏览器。或者，修改 curl 模块。使其可以支持回调函数，当并发请求中一个请求完成

时，立刻调用回调函数。这两种方法目前还在探索中。

6.2 直接调用 flush 函数输出

到这里，可能会有这样的疑问，为什服务器不直接把生成好的 HTML 内容分部 flush() 返回给客户端，而是使用 json 格式传递，然后用 js 解析呢？这不是多此一举么？实际上，这也是目前主搜索前端使用的方法。我们看看使用 BigPipe 方式的两大好处：

(1) 如果直接调用 flush()函数输出 html 源码，当模块较多的情况，模块间必须按顺序加载，在 html 前面的模块必须先加载完，后面的才能加载，这样也就没办法每个模块同时显示一些内容。例如下面的 html：

上面 3 个 div 分别代表 3 个模块，如果直接分部输出 html ，服务器端必须先加载完毕 div1 模块中的内容并 flush 出去后，才能继续加载 div2 的内容,如果 flush 顺序不一样，输出的 html 结构肯定就会出问题，这样就导致前台页面没办法同时显示 3 个 loading。因为这样 flush 必须要有先后顺序。而如果采用 JS 的话，可以前台显示 3 个 loading，而且不需要关心到底哪个模块先加载完，这样还能发挥后台多线程处理数据的优势。

(2)使用 JS 这种方式可以是页面结构更加清晰，管理更加方便。同时做到了页面逻辑结构和数据解耦，首先返回的是页面的结构，接着不断地返回 js 脚本，然后动态添加页面内容，而不是所有完整的 html 源码一起输出，增加了可维护性。

6.3 访问者是爬虫或者访问者浏览器禁止使用 JS 的情况

我们知道 BigPipe 使用 js 脚本加载页面，那么当用户在浏览器里设置禁止使用 js 脚本（虽然人数很少），就会造成加载页面失败，这同样是非常不好的用户体验。对搜索引擎的爬虫来讲，同样会遇到类似的问题。解决办法是当用户发送访问请求时，服务器端检测 user-agent 和客户端是否支持 js 脚本。如果 user-agent 显示是一个搜索引擎爬虫或者客户端不支持 js，就不使用 BigPipe ，而用原有的模式，从而解决问题。

6.4 对 SEO 的影响

这是一个必须考虑的问题，如今是搜索引擎的时代，如果网页对搜索引擎不友好，或者使搜索引擎很难识别内容，那么会降低网页在搜索引擎中的排名，直

接减少网站的访问次数。在 BigPipe 中，页面的内容都是动态添加的，所以可能会使搜索引擎无法识别。但是正如前面所说，在服务器端首先要根据 user-agent 判断客户端是否是搜索引擎的爬虫，如果是的话，则转化为原有的模式，而不是动态添加。这样就解决了对搜索引擎的不友好。

6.5 融合其他技术

除了使用 BigPipe，Facebook 的页面加载技术还融合了其他的页面优化技术，具体如下：

6.5.1 资源文件的 G-zip 压缩

这是非常重要的技术，使用 G-zip 对 css 和 js 文件压缩可以使大小减少 70%，这是多么诱人的数字！在网络传输的文件中，主要就是样式表和脚本文件。如此可以大大减小传输的内容，使页面加载速度变得更快。具体实现可以借助服务器来进行，例如 Apache，使用 mod_deflate 模块来完成具体配置为：

AddOutputFilterByType DEFLATE text/html text/css application/xjavascript

6.5.2 将 js 文件进行了精简

对 js 文件进行精简，可以从代码中移除不必要的字符，注释以及空行以减小 js 文件的大小，从而改善加载的页面的时间。精简 js 脚本的工具可以使用 JSMin，使用精简后的脚本的大小会减少 20%左右。这也是一个很大的提升。

6.5.3 将 css 和 js 文件进行合并

这是前端优化的一项原则，将多个样式表和 js 文件进行合并，这样的话，将会减少 http 的请求个数。对于上亿用户的网站来说，这也会带来性能的提升，大约会减少 5%左右的时间损耗。

6.5.4 使用外部 JS 和 CSS

同样是前端优化的一项原则。纯粹就速度来言，使用内联的 js 和 css 速度要更快，因为减少了 http 请求。但是，使用外部的文件更有利于文件的复用，这与面向对象编程的概念很像。更为重要的是，虽然在第一次的加载速度慢一点，但 css 文件和 js 脚本是可以被浏览器缓存。即之后用户的多次访问中，使用外部的 js 和 css 将会将会更好的提升速度。

6.5.5 将样式表放在顶部

和上面内容相似，这也是一种规范，将 html 内容所需的 css 文件放在首部

加载是非常重要的。如果放在页面尾部，虽然会使页面内容更快的加载（因为将加载 css 文件的时间放在最后，从而使页面内容先显示出来），但是这样的内容是没有使用样式表的，在 css 文件加载进来后，浏览器会对其使用样式表，即再次改变页面的内容和样式，称之为“无样式内容的闪烁”，这对于用户来说当然是不友好的。实现的时候将 css 文件放在<head>标签中即可。

6.5.6 将脚本放在底部实现“barrier”

支持页面动态内容的 Js 脚本对于页面的加载并没有什么作用，把它放在顶部加载只会使页面更慢的加载，这点和前面的提到的 css 文件刚好相反，所以可以将它放在页尾加载。是用户能看到的页面内容先加载，js 文件最后加载，这样会使用户觉得页面速度更快。Bigpipe 实现一个“barrier”的概念，即当所有的 pagelet 的内容全部加载好了之后，浏览器再向服务器发送 js 的 http 请求。可以在 BigPipe.js 中将所有的 pagelet 所需的 js 文件的路径保存下来，在判断所有的内容加载完成后统一向服务器发送请求。

7 BigPipe 具体实现细节

如上文讨论的那样，具体实现如下：当用户访问该页面时，在第一个 flush 的 Response 内容中，返回大部分的 HTML 代码，包括完整的<heaad>标签，和一个未封闭的<body>，其中<head>标签中有需要导入的文件的路径，如一些公共的 css 文件和 BigPipe.js 文件，<body>标签有页面的主要布局，第二块 flush 的内容为一段 js 脚本，处理 BigPipe 对象的生成，以及 js 和 css 文件的路径和字符串的映射

```
var bigPipe = new bigPipe();
bigPipe.setResourceMap({
aaaaa:{
“name”: “js/list1.js”,
“type”: “js”,
“src”: “js/list1.js”
}
);
```

setResourceMap(json)为 BigPipe 中的函数，功能是设置文件的映射。”aaaaa”

应该是在服务器随即生成的五位字符串,name 表示文件名称，type 为文件的类型，可以是”js”或”css”，”src”为文件的路径。在下面的页面中，就可以使用”aaaaa”来替代”js/list1.js”了，减少了复杂性。接下来 flush 的是每一个 pagelet 的内容了，例如：

```
<script type=”text/javascript” >
bigPipe.onPageletArrive({
id:”list1”,
content:”this is list 1 <\/br><img src =\”img13.jpg\” \/>”,
css:["eeeee"],
js:["aaaaa"],
“resource_map”:{
aaaaa:{
“name”: “js/list1.js”,
“type”: “js”,
“src”: “js/list1.js”
},
“eeeee”: {
“name”: “css/list1.css”,
“type”: “css”
“src”: “css/list1.css”
}
}
});
</script>
```

onPageletArrive（json_arrive）也是 BigPipe 的函数，功能是动态添加页面的内容和加载 pagelet 所需的文件，函数的参数为 json 格式的数据。其参数含义是：“id”用来寻找 pagelet 标签；“ content ”是 html 页面内容，在找到对应的 pagelet 的标签之后，将 content 内动态添加到 html 页面中；“css”为该 Pagelet 所需的 css 文件，这里的 css 文件可能在之前导入过了；“js”为该 pagelet 所需的

js 文件，同样，有可能在之前的 pagelet 已经导入过了。在函数实现过程中，因为 js 文件是最后加载的，可以把这些 js 的路径存入到一个数组当中（去掉重复的），在最后一起加载。resource_map”为该 pagelet 所单独需要加载的 js 和 css 文件，同样也是 json 格式的，结构与前面的 setResource()中的参数一样。最后 flush 的是

</body>

</html>

即为最后的标签。

8 结论

经过上面的讨论，我们可以发现，使用 BigPipe 技术优化页面可以有四个好处：

1. 减少页面的加载时间
2. 使页面分步输出，改善用户体验
3. 使页面结构化，提高可读性，更加便于维护
4. 每个 pagelet 都是相互独立的，如果有一个 pagelet 的内容不能加载，并不会影响其他的 pagelet 的内容显示。

同时，BigPipe 是一项比较新的理念， 在去年六月份才由 Facebook 的工程师提出，应该说有很大的发展空间。BigPipe 的原理非常简单，并不会引入很多额外的负担，适用范围很广，容易上手。几乎所有的网页都可以采用 BigPipe 的理念去进行优化，尤其对于是有着海量数据和网页比较大的网站，将会以低成本带来高回报。一般来讲，网站越大，脚本和样式表越多，浏览器版本越旧，网络环境越差，优化的结果越可观。

原文链接：http://blog.sae.sina.com.cn/archives/2784

Using Bootstrap 3 With Sass

Bootstrap comes with Responsive Grids, and a few common web components that we can pick up to build a responsive website quickly. If you have seen our previous posts on Bootstrap, you probably know that Bootstrap styles are composed using LESS.

While LESS has become more powerful with new features introduced in version 1.5, some of you might be more familiar with Sass/SCSS. There may also be some features in Sass that you can't live without, but they are not present in LESS (yet). If you want to work on Boostrap and with Sass, thanks to Thomas McDonald, you can because Bootstrap has been ported to Sass/SCSS.

Recommended Reading: A Look Into: Foundation 4 Responsive Framework

Installation

There are a few ways to start using Bootstrap + Sass. First, since it has been included as a Ruby gem, you can install it through Terminal with the following command line:

gem install bootstrap-sass

You can also use it along with Compass with this command below. It's the same way as how we install Zurb Foundation. But, please note that this way will only include the _variables.lesscontaining Bootstrap variables, and styles.less where you

put your own style-rules.

compass create my-new-project -r bootstrap-sass --using bootstrap

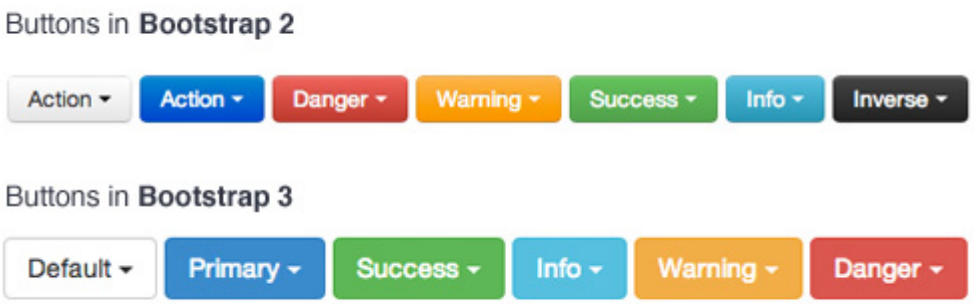
Alternatively, you can simply download it from the Github repo.

What's New In Bootstrap 3

Here are a few new features found in Bootstrap 3.

Flat Design

The change that you can immediately see from the new version, Bootstrap 3, is that it is now embraces flat design. The gradients, and shadows that we found in the previous version components are now gone.



Grid In Bootstrap 3

Bootstrap also introduces a set of new classes and new grid constructions. In version 3, there are Large, Medium, Small, and Extra Small Grids to cater to different viewport sizes. Let's see the following HTML example:

view plaincopy to clipboardprint?

```
<div class="container">
    <div class="row">
        <div class="column col-md-4 col-sm-6">
            <p>Left Column</p>
        </div>
        <div class="column col-md-4 col-sm-4">
            <p>Middle Column</p>
        </div>
        <div class="column col-md-4 col-sm-2">
```

```
                <p>Right Column</p>
            </div>
        </div>
    </div>
```

We have three columns. Each column has an equal width when viewed in a large viewport size (on a desktop screen or landscape orientation on the tablet). The size is applied with col-md-4 class.

Then, when the screen size is getting smaller the column width division will be adjusted with thecol-sm-* class, so that column width could remain in the right proportion rather than just being stacked, like in the previous version of Bootstrap.

New Components

There are also some new Components added in version 3. This includes Pager (used for building Next-and-Prev type of navigation), List Group, Panels, and Page Header.

Utilizing Sass Functions

Technically, we can just add Bootstrap classes to the HTML elements to make the website layout, as we did in the example above. But, when using CSS Pre-processors, like Sass, we can utilize some of the functions to achieve a cleaner and more semantic HTML structure rather than being stuffed with meaningless class names.

Given the previous example, we can change the structure as well as class names to something like this:

view plaincopy to clipboardprint?

```
<div class="container">
    <div class="main-area">
        <div class="column content">
            <p>This is the Content.</p>
        </div>
        <div class="column sidebar">
            <p>This is the Sidebar.</p>
        </div>
        <div class="column side-nav">
            <p>This is the Navigation.</p>
        </div>
    </div>
</div>
```

Within the stylesheet, we can use Sass @extend directive to build the layout. Using @extend will group the selectors that share the same style-rules.

view plaincopy to clipboardprint?

```
.main-area {
    @extend .row;
}
.column {
    @extend .col-md-4;
}
.content {
    @extend .col-xs-6;
    @extend .col-sm-6;
}
.sidebar {
    @extend .col-xs-4;
    @extend .col-sm-4;
```

```
}
.side-nav {
    @extend .col-sm-2;
    @extend .col-sm-2;
}
```

Alternatively, you can also use Sass @include which will copy and include the style-rules from mixins into our class selectors.

view plaincopy to clipboardprint?

```
.main-area {
    @include make-row;
}
.content {
    @include make-xs-column(6);
    @include make-sm-column(6);
}
.sidebar {
    @include make-xs-column(4);
    @include make-sm-column(4);
}
.side-nav {
    @include make-xs-column(2);
    @include make-sm-column(2);
}
.column {
    @include make-md-column(4);
}
```

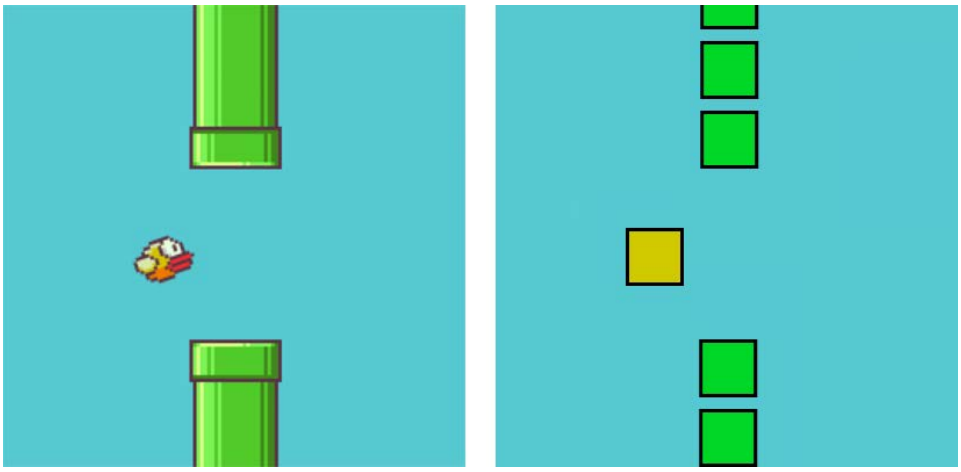
Now, view it on the browser and you will get your responsive layout.

原文链接：http://www.hongkiat.com/blog/bootstrap-and-sass/

How to make a Flappy Bird in HTML5 with Phaser

Flappy Bird is a nice little game with easy to understand mechanics, and I thought it would be a perfect fit for an HTML5 game tutorial. So in this tutorial we are going to make a simplified version of Flappy Bird, in only 65 lines of Javascript.

Click here to test the game we are going to build.



Note 1: you need to know some Javascript in order to understand this tutorial.

Note 2: If you want to learn more about the Phaser framework and how to set up your environment, you should read my short getting started tutorial first.

Set up

You should download this basic template that I made for this tutorial. In it you will find:

phaser.min.js, the minified Phaser framework v1.1.5.

index.html, where the game will be displayed.

main.js, a file where we will write all the code.

assets/, a directory with 2 images (bird.png and pipe.png).

Put all of these files on your webserver. Open the index.html file in your browser, and open the main.js file in your text editor.

In the main.js file you should see the basic structure of a Phaser project that we discussed in the previous tutorial.

```
// Initialize Phaser, and creates a 400x490px gamevar game = new Phaser.Game(400, 490, Phaser.AUTO, 'game_div');
var game_state = {};
// Creates a new 'main' state that will contain the gamegame_state.main = function() { };
game_state.main.prototype = {
    preload: function() {
        // Function called first to load all the assets    },
    create: function() {
        // Fuction called after 'preload' to setup the game        },
    update: function() {        // Function called 60 times per second    },};
// Add and start the 'main' state to start the gamegame.state.add('main', game_state.main);
game.state.start('main');
```

We are going to fill in the preload(), create() and update() functions, and add some new functions to make the game work.

The bird

Okay, so now you are ready to code! Let's first focus on adding a bird to the game that can be controlled with the spacebar key.

Everything is quite simple and well commented, so you should be able to understand easily the code below. For better readability, I removed the Phaser initialization and states management code that you can see above.

First we update the preload(), create() and update() functions.

```
    preload: function() {
        // Change the background color of the game
this.game.stage.backgroundColor = '#71c5cf';
        // Load the bird sprite    this.game.load.image('bird',
'assets/bird.png');
    },
    create: function() {
        // Display the bird on the screen    this.bird =
this.game.add.sprite(100, 245, 'bird');
        // Add gravity to the bird to make it fall
this.bird.body.gravity.y = 1000;
        // Call the 'jump' function when the spacekey is hit    var
space_key = this.game.input.keyboard.addKey(Phaser.Keyboard.SPACEBAR);
space_key.onDown.add(this.jump, this);
    },
    update: function() {
        // If the bird is out of the world (too high or too low), call
the 'restart_game' function    if (this.bird.inWorld == false)
this.restart_game();},
```

And just below this, add these two new functions.

```
    // Make the bird jump jump: function() {
        // Add a vertical velocity to the bird
this.bird.body.velocity.y = -350;},
    // Restart the gamerestart_game: function() {
        // Start the 'main' state, which restarts the game
this.game.state.start('main');},
```

Save the main.js file with the new code, and reload the index.html file. You should have a bird jumping when you press the spacebar key.

The pipes

A Flappy Bird game without pipes is not really interesting, so let's change that.

First, load the pipe sprite in the preload() function.

```
this.game.load.image('pipe', 'assets/pipe.png');
```

Then, create a group of pipes in the create() function. Since we are going to handle a lot of pipes in the game, it's easier to use a group of sprites. The group will contain 20 sprites that we will be able to use as we want.

```
this.pipes = game.add.group();
this.pipes.createMultiple(20, 'pipe');
```

Now we need a new function to add a pipe in the game. By default, all the pipes contained in the group are dead and not displayed. So we pick a dead pipe, display it, and make sure to automatically kill it when it's no longer visible. This way we never run out of pipes.

```
add_one_pipe: function(x, y) {
    // Get the first dead pipe of our group     var pipe =
this.pipes.getFirstDead();
    // Set the new position of the pipe     pipe.reset(x, y);
    // Add velocity to the pipe to make it move left
pipe.body.velocity.x = -200;
    // Kill the pipe when it's no longer visible
pipe.outOfBoundsKill = true;},
```

The previous function displays one pipe, but we need to display 6 pipes in a row with a hole somewhere in the middle. So let's create a new function that does just that.

```
add_row_of_pipes: function() {
    var hole = Math.floor(Math.random()*5)+1;
    for (var i = 0; i < 8; i++)          if (i != hole && i != hole +1)
            this.add_one_pipe(400, i*60+10);
```

},

To actually add pipes in the game, we need to call the add_row_of_pipes()function every 1.5 seconds. We can do this by adding a timer in the create() function.

this.timer = this.game.time.events.loop(1500, this.add_row_of_pipes, this);

And finally add this line of code a the beginning of the restart_game()function to stop the timer when we restart the game.

this.game.time.events.remove(this.timer);

Scoring and collisions

The last thing we need to finish the game is adding a score and handling collisions. And this is quite easy to do.

Add this in the create() function to display a score label in the top left.

this.score = 0;

var style = { font: "30px Arial", fill: "#ffffff" };

this.label_score = this.game.add.text(20, 20, "0", style);

Put this in the add_row_of_pipes(), to increase the score by 1 each time new pipes are created.

this.score += 1;

this.label_score.content = this.score;

And add this in the update() function to call restart_game() each time the bird collides with a pipe.

this.game.physics.overlap(this.bird, this.pipes, this.restart_game, null, this);

And we are done! Congratulation, you now have an HTML5 Flappy Bird clone. You can download the full source code here.

原文链接：http://blog.lessmilk.com/how-to-make-flappy-bird-in-html5-1/

[编程语言]

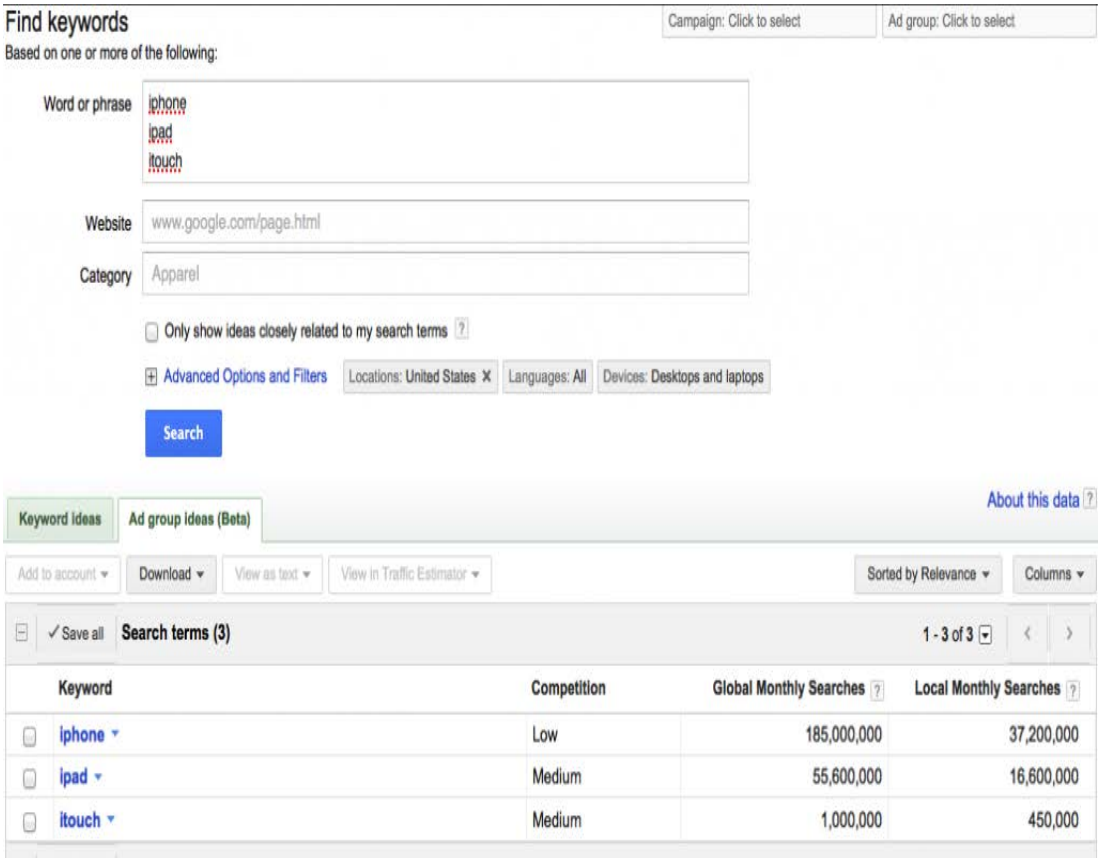
使用 python/casperjs 编写终极爬虫-客户端 App 的抓取

1.缘起

随着移动互联网的发展，现在写 web 和我三年前刚开始写爬虫的时候已经改变了太多。特别是在 node 以及 javascript/ruby 社区的努力下，以往“服务器端”做的事情都慢慢搬到了“浏览器”来实现，最极端的例子可能是 meteor 了 ，写 web 程序无需划分前端后端的时代已经到来了。。。

在这一方面，Google 一向是最激进的。纵观 Google 目前的产品线，社交的 Google Plus，网站分析的 Google Analytics，Google 目前赖以生存的 Google Adwords 等，如果想下载源码，用 ElementTree 来解析网页，那什么都得不到，因为 Google 的数据都是通过 Ajax 调用经过数据混淆处理的数据，然后用 JavaScript 进行解析渲染到页面上的。

本来这种事情也不算太多，忍一忍就行了，不过最近因业务需要，经常需要上 Google 的 Keyword Tools 来分析特定关键字的搜索量。



图为关键字搜索的截图

图为 Google 经过混淆处理的 Ajax 返回结果。

要把这么费劲的事情自动化倒也不难，因为 Google 也提供了 API 来做，Adwords 项目的 TargetingIdeaService 就是来做这个的，问题是 Google 的 API 调用需要花钱，而如果能用爬虫技术来爬取这个结果，就能省去不必要的额外开销。

2. Selenium WebDriver

由于要解析执行复杂的 JavaScript，必须有一个 Full Stack 的浏览器 JavaScript 环境，这种环境三年前的话，可能只能诉诸于于 selenium，selenium 是一款多语言的浏览器 Driver，它最大的优点在于，提供了从命令行统一操控多种不同浏览器的方法，这极大地方便了 web 产品的兼容性自动化测试。

2.1 在没有图形界面的服务器上安装和使用 Selenium

安装 selenium 非常简单，pip install selenium 即可，但是要让 firefox/chrome 工作，没有图形界面的话，还是要费一番功夫的。

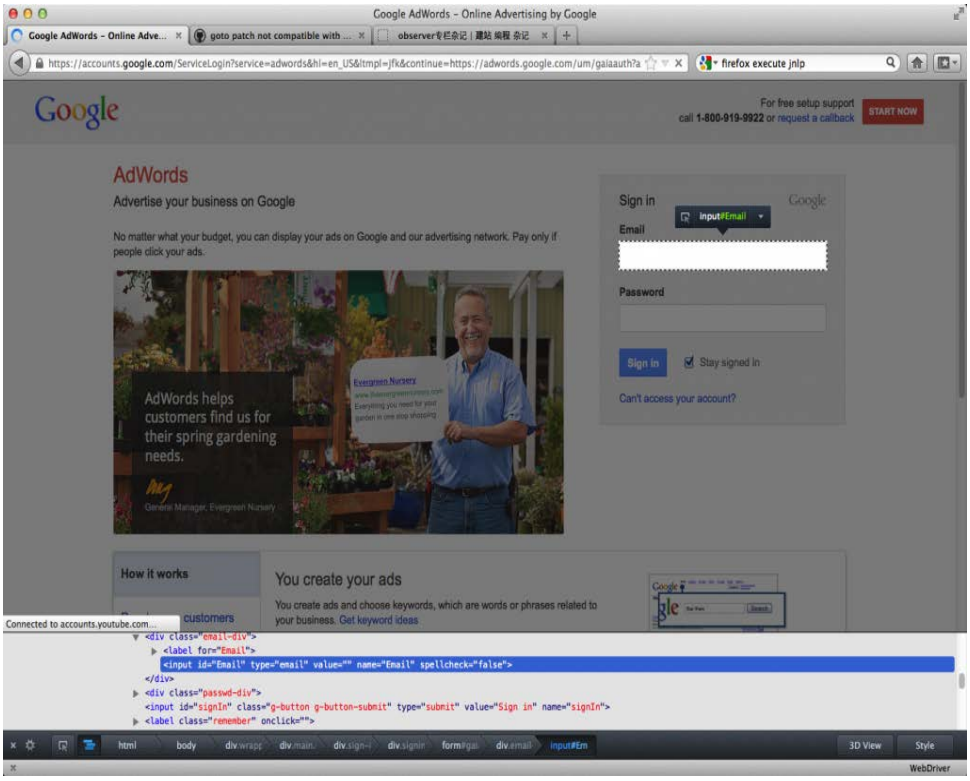
推荐的做法是

```
1    apt-get install xvfb
2    Xvfb :99 -ac -screen 0 1024x768x8&amp;amp;
3    export DISPLAY=:99
```

Selenium 的安装和配置在此就不多说了，值得注意的是，如果是 Ubuntu 用户，并且要使用 Chrome 的话，必须额外下载一个 chromedriver，并且把安装的 chromium-browser 链接到/usr/bin/google-chrome，否则将无法运行。

2.2 爬取 Keywords

先总结一下 Adwords 的使用方法吧，要能正常使用 Adwords，必须要有一个开通 Adwords 的 Google Account，这倒不是很难，只要访问 http://adwords.google.com ，Google 会协助创建账号，如果还没有的话，其次就是登陆了。



通过分析登陆页面，我们可以看到需要在 id 为 Email 的一个 input 框内输入 email，然后在 id 为 Passwd 的密码框内输入密码，然后点击 Sign in 提交这个页面内唯一的 form。

首先别忘了开一个浏览器先

```
1        driver.find_element_by_id("Email").send_keys(email)
2        driver.find_element_by_id("Passwd").send_keys(passwd)
3        driver.find_element_by_id('signIn').submit()
```

登陆后，我们发现需要访问一个类似 https://adwords.google.com/o/Targeting/Explorer 的网页才能跳转到关键字工具，于是我们手动生成一下这个网页

到了工具主页以后，事情就变得 Tricky 起来了。因为整个关键字工具都是个客户端 App，在全部文件载入完成以后，页面不会直接渲染完毕，而是要经过复

```
1                driver.implicitly_wait(30)
```

杂的 JavaScript 运算后页面才会完整显示。然而 Selenium WebDriver 并不知道这一点，所以我们要让他知道。

在这里，我们要等待 Search 按钮在浏览器中出现以后，才能确认网页加载完毕，Selenium WebDriver 有两种方式可以实现这一点，我偷懒用了全局的默认等待机制：

于是 Selenium 就会在找不到页面元素的时候自动等候不超过 30 秒

接下来，等待输入框和 Search 按钮出现后提交搜索 iphone 关键字的请求

```
1        driver.find_element_by_class_name("sEAB").send_keys("iphone")
2        find_element_by_css_selector("button.gwt-Button").click()
```

然后我们继续等待 class 为 sLNB 的 table 的出现，并解析结果

```
result = {}
texts = driver.find_elements_by_xpath('//table[@class="sLNB"]')\
                [0].text.split()
for i in range(1, len(texts)/4):
        result[ texts[i*4] ] = (texts[i*4+2], texts[i*4+3])
```

这里我们使用了 xpath 来提取网页特征，这也算是写爬虫的必备吧。

完整的例子见： https://gist.github.com/3798896 替换 email 和 passwd 后直接就能用了

3. JavaScript Headless 解决方案

随着 Node 以及随之而来的 JavaScript 社区的进化，如今的我们就幸福多了。远的我们有 phantomjs，一个 Headless 的 WebKit Driver，意味着可以无需 GUI，完全模拟 Chrome/Safari 的操作。近的有 casperjs（基于 phantomjs 的好用封装），zombie（相比 phantomjs 的优势是可以和 node 集成）等。

其中非常可惜地是，zombiejs 似乎对富 JavaScript 网站支持得有问题，所以后来我还是只能用 casperjs 来进行测试。Headless 的方案因为不需要渲染 GUI，执行速度约为 Selenium 方案的三倍。

另外由于这是纯 JavaScript 的方案，于是我们可以直接在例如 Chrome 的 Console 模式下写代码控制浏览器，不存在如 Selenium 那样还需要语义转换，非常简洁直观。例如利用 W3C Selectors API Level 1 所提供的 querySelector 来快速选取元素，对表单进行 submit，对按钮进行 click，甚至可以执行自定义 JavaScript 脚本以便按一定规律对页面进行操控。

但是 casperjs 或者说 phantomjs 的弱点是不支持除了文件读写和浏览器操作以外的一切*nix IPC 惯用伎俩，socket 神马的统统不支持，1.4 版本以后才加入了一个 webserver 用于和外界通信，但是用 httpserver 来和外界通信？我有点抵触就是了。

废话不说了，casperjs 的代码看起来就是这样，登陆

```
var casper = require('casper').create({verbose:true,logLevel:"debug"});
        casper.start('http://adwords.google.com');
        casper.thenEvaluate(function login(email, passwd) {
        document.querySelector('#Email').setAttribute('value', email);
        document.querySelector('#Passwd').setAttribute('value', passwd);
        document.querySelector('form').submit();
        }, {email:email, passwd:passwd});

        casper.waitForSelector(".aw-cues-item", function() {
        kwurl = this.evaluate(function(){
        var search = document.location.search;
        return
'https://adwords.google.com/o/Targeting/Explorer'+search+'&amp;__o=cues&
amp;ideaRequestType=KEYWORD_IDEAS';
        })
        });
```

与 Selenium 类似，因为页面都是 Ajax 调用的，我们需要明确地“等待某个元素出现”，即：waitForSelector，casperjs 的文档既简洁又漂亮，不妨多逛逛。

值得一提的是，casperjs 一定要调用 casper.run 方法，之前的 start, then 等方法，只是把步骤封装到了 this._steps 里面，只有在 run 的时候才会真正执行，所以 casperjs 设计流程的时候会很痛苦，for/each 之类的手法有时并不好用。

这个时候需要用 JavaScript 编程比较常用的递归化的方法，参见 https://github.com/n1k0/casperjs/blob/master/samples/dynamic.js 这个例子。我在

完整的 casperjs 代码里面也是这么做的。

具体逻辑的实现和 selenium 类似，我就不废话了，完整的例子参见：https://gist.github.com/3798922

4. 综上

介绍了 selenium 和 casperjs 两种不同的终极爬虫写法，但是其实这篇文写来只是太久没更新了，写点东西更新一下而已：）

原文链接：http://blog.sae.sina.com.cn/archives/2763

2013 年度 Python 运维工具

Pycoders 周刊根据读者对周刊文章的点击数据，评选出了 2013 年最受关注的 Python 运维工具。

metrology (github.com)

这个库很酷，支持你对应用进行多种测量，并轻松的输出给类似 graphite 的外部系统.

python-lust (github.com)

支持在 Unix 系统中用 Python 实现一个守护进程.

scales (github.com)

Scales 对你的 Python 应用进行持续状态和统计，并发送数据到 graphite. 详情/实例 查阅官方 README.

glances (github.com)

跨平台基于 curses 命令行的系统监视工具.

(译注:htop 的纯 Python 替代，大妈已经用上;-)

ramona (github.com)

企业级的应用监管. Ramona 保证每个进程在值, 一但需要立即重启, 并有监控/日志输出, 查觉要糟时,会发送邮件提醒.

salmon (github.com)

Salmon 是基于 Salt Stack 的多服务监视系统. 即能作报警系统,也能当监控系统, README 有截屏以及详细说明.

graph-explorer (github.com)

Graph-explorer 是对 Graphite 面板的增强. 比原版的好很多,值得体验.

sovereign (github.com)

Sovereign 是一系列 ansible 的攻略手册, 基于之,能为自个儿建造个私人云.

(译注: Ansible, Fabric, SlatStack, 这是 Python 实现的类似 Puppet 的持续部署管理系统 但是,更加简洁,直觉,值得关注)

shipyard (github.com)

(shipyard,船坞) 名很倩的 web 应用, 可以显示给定机器上的 docker 实例. 也支持创建、删除等操作.

docker-py (github.com)

An API client for the amazing

疯狂的 docker 工程接口的 Python 包装.

(译注: 不没听说过 Docker? 忒 out 了,参考: 无责任报道~ECUG2013Con; 这是准备将应用部署连操作系统环境也一并抄底儿统一快照/回滚/分发/版本管理的系统)

dockerui (github.com)

基于 docker 接口通过 web 界面进行交互操作的工具.

django-docker (github.com)

如果想知道怎么将 Djnago 应用同 Docker 结合? 学习这个 demo 吧.

diamond (github.com)

Python 实现的守护进程, 自动从你的服务或是其它指定数据源中提取数值, 并向 graphite 以及其它支持的 状态面板/收集 系统输出.

原文链接：http://segmentfault.com/a/1190000000413672

又被 Python 的 Unicode 坑了

很久没更新博客了，不是没什么可写，而是因为工作生活的种种原因没有心情更新。趁着今天有点时间，还是开个头继续写吧。

这次遇到的问题还是与 Python 的 Unicode 有关。请看下面的代码：

```
[~/tmp]$ cat test.py
#coding:utf8
foo = u'测试'
print foo
[~/tmp]$ python test.py
测试
[~/tmp]$ python test.py > /tmp/foobar.txt
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    print foo
UnicodeEncodeError: 'ascii' codec can't encode \
  characters in position 0-1: ordinal not in range(128)
```

简简单单打印一个 Unicode 对象，直接输出不会报错，但重定向到文件就会挂掉，这实在让人想“呵呵”。

一番 Google 之后在 Stackoverflow 上找到了答案。原来 Python 2 里的 print 会针对 unicode 参数进行自动 encode。如果输出的目标是一个终端，它会使用终端的编码（可通过 sys.stdout.encoding 获得）；如果输出的目标是管道（重定向或者使用管道符），则无法获取对应的编码，此时 Python 会使用系统默认的编码。

可以 print 一下 sys.stdout.encoding 查看终端编码：

```
[~/tmp]$ python -c 'import sys; print sys.stdout.encoding'
UTF-8
[~/tmp]$ python -c 'import sys; print sys.stdout.encoding' | tee /tmp/foo.txt
```

None

果然在使用管道的情况下，终端编码是 None 。

解决办法也很简单，就是在 print 之前手工 encode 一下，比如在 Linux 环境完全可以一律 encode 成 UTF-8。如果想做得更好一点，也可以根据 sys.stdout.encoding 以及是否有重定向来判断决定。

刚开始写博客的时候就遇到过 Unicode 格式化的坑，没想到快两年后还会再遇到类似问题。不得不说 Python 2 的默认编码，以及 Unicode 和 Str 这块容易出问题。可惜的是，虽然 Python 3 完美地解决了这些问题，但都 3.X 了，还是没能大面积普及起来，真是遗憾。

原文链接：http://jerrypeng.me/2014/02/python-2-unicode-print-pitfall/

FutureTask 源码解析

站在使用者的角度，future 是一个经常在多线程环境下使用的 Runnable，使用它的好处有两个：

1. 线程执行结果带有返回值
2. 提供了一个线程超时的功能，超过超时时间抛出异常后返回。

那，怎么实现 future 这种超时控制呢？来看看代码：

```
public class FutureTask<V> implements RunnableFuture<V> {
    /** Synchronization control for FutureTask */
    private final Sync sync;
```

FutureTask 的实现只是依赖了一个内部类 Sync 实现的，Sync 是 AQS (AbstractQueuedSynchronizer)的子类，这个类承担了所有 future 的功能，

AbstractQueuedSynchronizer 的作者是大名鼎鼎的并发编程大师 Doug Lea，它的作用远远不止实现一个 Future 这么简单，后面在说。

下面，我们从一个 future 提交到线程池开始，直到 future 超时或者执行结束来看看 future 都做了些什么。怎么做的。

首先，向线程池 ThreadPoolExecutor 提交一个 future：

```
future = exec.submit( new WebDivideFuture(cookieUtils,
                                           jedisUtil,
                                           request,
                                           selectFactory,
                                           result,
                                           testInfos) );
```

ThreadPoolExecutor 将提交的任务用 FutureTask 包装一下：

```
public <T> Future<T> submit(Callable<T> task) {
    if (task == null) throw new NullPointerException();
    RunnableFuture<T> ftask = newTaskFor(task);
    execute(ftask);
    return ftask;
}
```

```
protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {
    return new FutureTask<T>(callable);
}
```

然后尝试将包装后的 Future 用 Thread 类包装下后启动，

红色标记的地方表示，当当前线程池的大小小于 corePoolSize 时，将任务提交，否则将该任务加入到 workQueue 中去，如果 workQueue 装满了，则尝试在线程数小于 MaxPoolSize 的条件下提交该任务。

```
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    if (poolSize >= corePoolSize || !addIfUnderCorePoolSize(command)) {
        if (runState == RUNNING && workQueue.offer(command)) {
            if (runState != RUNNING || poolSize == 0)
                ensureQueuedTaskHandled(command);
        }
        else if (!addIfUnderMaximumPoolSize(command))
            reject(command); // is shutdown or saturated
    }
}
```

顺便说明下，我们使用线程池时，常常看到有关有界队列，无界队列作为工作队列的字眼：使用无界队列时，线程池的大小永远不大于 corePoolSize，使

用有界队列时的 maxPoolSize 才有效，原因就在这里，如果是无界队列，红框中的 add 永远为 true 下方的 addIfUnderMaximumPoolSize 怎么也走不到了，也就不会有线程数量大于 MaxPoolSize 的情况。

言归正传，看看 addIfUnderCorePoolSize 中做了什么事：new 了一个 Thread，将我们提交的任务包装下后就直接启动了

```
private boolean addIfUnderCorePoolSize(Runnable firstTask) {
    Thread t = null;
    final ReentrantLock mainLock = this.mainLock;
    mainLock.lock();
    try {
        if (poolSize < corePoolSize && runState == RUNNING)
            t = addThread(firstTask);
    } finally {
        mainLock.unlock();
    }
    if (t == null)
        return false;
    t.start();
    return true;
}
```

我们知道，线程的 start 方法会调用我们 runnable 接口的 run 方法，因此不难猜测 FutureTask 也是实现了 Runnable 接口的

```
 */
public class FutureTask<V> implements RunnableFuture<V> {
    /** Synchronization control for FutureTask */
```

```
public interface RunnableFuture<V> extends Runnable, Future<V> {
    /**
     * Sets this Future to the result of its computation
     * unless it has been cancelled.
     */
    void run();
}
```

FutureTask 的 run()方法中是这么写：

```
public void run() {
    sync.innerRun();
}
```

innerRun 方法先使用原子方式更改了一下自己的一个标志位 state(用于标

示任务的执行情况）

然后红色框的方法 实现回调函数 call 的调用，并且将返回值作为参数传递下去，放置在一个叫做 result 的泛型变量中，

然后 future 只管等待一段时间后去拿 result 这个变量的值就可以了。 至于怎么实现的“等待一段时间再去拿” 后面马上说明。

innerSet 在经过一

```
void innerRun() {
    if (!compareAndSetState(0, RUNNING))
        return;
    try {
        runner = Thread.currentThread();
        if (getState() == RUNNING) // recheck after setting thread
            innerSet(callable.call());
        else
            releaseShared(0); // cancel
    } catch (Throwable ex) {
        innerSetException(ex);
    }
}
```

系列的状态判断后，最终将 V 这个 call 方法返回的值赋值给了 result

```
void innerSet(V v) {
for (;;) {
int s = getState();
if (s == RAN)
    return;
        if (s == CANCELLED) {
    // aggressively release to set runner to null,
    // in case we are racing with a cancel request
    // that will try to interrupt runner
            releaseShared(0);
            return;
        }
if (compareAndSetState(s, RAN)) {
            result = v;
            releaseShared(0);
            done();
    return;
        }
    }
}
```

说到这里，我们知道，future 是通过将 call 方法的返回值放在一个叫做 result 的变量中，经过一段时间的等待后再去拿出来返回就可以了。

怎么实现这个 “等一段时间”呢？

要从 Sync 的父类 AbstractQueuedSynchronizer 这个类说起：

我们知道 AbstractQueuedSynchronizer 后者的中文名字叫做 同步器，顾名思义，是用来控制资源占用的一种方式。对于 FutureTask 来说，“资源”就是 result，线程执行的结果。思路就是通过控制对 result 这个资源的访问来决定是否需要马上去取得 result 这个结果，当超时时间未到，或者线程未执行结束时，是不能去取 result 的。当线程正常执行结束后，一系列的标志位会被修改，并告诉等待 future 执行结果的各个线程，可以来获取 result 了。

这里会涉及到 独占锁和共享锁的概念。

独占锁：同一时间只有一个线程获取锁。再有线程尝试加锁，将失败。 典型例子 reentrantLock

共享锁：同一时间可以有多个线程获取锁。 典型例子，本例中的 FutureTask

为什么说他们？因为 Sync 本质上就是想完成一个共享锁的功能，所以 Sync 继承了 AbstractQueuedSynchronizer 所以 Sync 的方法使用的是 AbstractQueuedSynchronizer 的共享锁的 API

首先，我们明白，future 结束有两种状态：

1. 线程正常执行完毕，通知等待结果的主线程对应于 future.get()方法。
2. 线程还未执行完毕，等待结果的主线程已经等不到了（超时），抛出一个 TimeOutException 后不再等待。对应于 future.get(long timeout, TimeUnit unit)

下面我们依次看看对于这两种状态，我们是怎么处理的：

从上图中可以得知，线程在执行完毕后会将执行的结果放到 result 中， 红色框中同时提到了 releaseShared 方法，我们从这里进入 AbstractQueuedSynchronizer

```
void innerSet(V v) {
for (;;) {
int s = getState();
if (s == RAN)
    return;
        if (s == CANCELLED) {
    // aggressively release to set runner to null,
    // in case we are racing with a cancel request
    // that will try to interrupt runner
            releaseShared(0);
            return;
        }
if (compareAndSetState(s, RAN)) {
            result = v;
            releaseShared(0);
            done();
    return;
        }
    }
}
```

当 result 已经被赋值，或者 FutureTask 为 cancel 状态时，FutureTask 会尝试去释放共享锁（可以同时有多个线程调用 future.get() 方法，也就是会有多个线程在等待 future 执行结果,而 furue 在执行完毕后会依次唤醒各个线程）如果尝试成功，则开始真正的释放锁，这里是 AbstractQueuedSynchronizer 比较精妙的地方， “尝试”动作都定义为抽象方法，交个各个子类去定义“尝试成功的含义” 而真正的释放则自己实现，这种复杂规则交个子类，流程交给自己的思路很值得借鉴。

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

再看 FutureTask 的 “尝试释放”的规则：

没啥好说，怎么尝试都成功

```
/**
 * Implements AQS base release to always signal after setting
 * final done status by nulling runner thread.
 */
protected boolean tryReleaseShared(int ignore) {
    runner = null;
    return true;
}
```

接着 AbstractQueuedSynchronizer 开始了真正的释放唤醒工作：

```
private void doReleaseShared() {

/*
* Ensure that a release propagates, even if there are other
* in-progress acquires/releases. This proceeds in the usual
* way of trying to unparkSuccessor of head if it needs
* signal. But if it does not, status is set to PROPAGATE to
* ensure that upon release, propagation continues.
* Additionally, we must loop in case a new node is added
* while we are doing this. Also, unlike other uses of
* unparkSuccessor, we need to know if CAS to reset status
* fails, if so rechecking.
*/
for (;;) {
     Node h = head;//把头元素取出来，保持头元素的引用，防止head被更改
     if (h != null &amp;amp;&amp;amp; h != tail) {
        int ws = h.waitStatus;
        if (ws == Node.SIGNAL) {//如果状态位为：需要一个信号去唤醒 注释原话：/** waitS
        if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0)) //修改状态位
             continue; // loop to recheck cases
        unparkSuccessor(h);//如果修改成功，则通过头元素找到一个线程，并且唤醒它（唤醒动作
        }
       else if (ws == 0 &amp;amp;&amp;amp;
             !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
       continue; // loop on failed CAS
     }
     if (h == head) // loop if head changed
         break;
   }
}
```

循环遍历后，知道已经没有结点需要唤醒则返回，依次 return 后，future 的 run 方法执行完毕。

以上是针对 future 线程的，我们知道，FutureTask 已经将执行结果放在了 result 中，并且按等的先后顺序依唤醒了等待队列上的线程。

那，猜测 future.get 方法就不难了，对于带超时的 get 方法：最大的可能性就是不断的检查 future 的一个状态位，看它是否执行完毕，执行完则获取结果返回，否则，再阻塞自己一段时间。

对于不待超时的，就上来就先尝试获取结果，拿不到就阻塞自己，直到上述的 innerSet 方法唤醒它。

究竟是不是这样呢？一起来看看：

因为 innerGet(long nanosTimeout) 和 innerGet()流程大致相同，所以我们重点讲解 innerGet(long nanosTimeout) ，在唯一一个有区别的地方说明下

即可。

如下图所示，对于 innerGet(long nanosTimeout) 方法，FutureTask 采用的方法是直接加锁或者每隔一段时间尝试加锁，如果成功，则返回 true，则如上图所示，直接返回 result，主线程拿到执行结果。

否则，抛出超时异常。

对于 tryAcquireShared 方法，比较简单，直接看 future 是否执行完毕

如果没有结束，则进入 doAcquireSharedNanos 方法：

```
private boolean doAcquireSharedNanos(int arg, long nanosTimeout)
throws InterruptedException {

    long lastTime = System.nanoTime();
    final Node node = addWaiter(Node.SHARED);//在队列尾部增加一个结点，我的理解是，用
    try {
        for (;;) {
            final Node p = node.predecessor();//拿出刚才新增结点的前一个结点：实际有效
            if (p == head) {
                int r = tryAcquireShared(arg);//尝试获取锁。
                if (r &gt;= 0) {//
                    setHeadAndPropagate(node, r);//返回值大于1 对于FutureTask代表任
                p.next = null; // help GC 将p结点脱离队列，帮助GC
            return true;//返回true后 上述中可以知道当前线成会抛出超时异常 确定下会不会唤
        }
        }
        if (nanosTimeout &lt;= 0) { //如果设置的超时时间小于等于0 则取消获取锁 cancelA
            shouldParkAfterFailedAcquire(p, node)) // 遍历队列，找到需要沉睡的第一个节
            LockSupport.parkNanos(this, nanosTimeout); // 调用JNI方法，沉睡当前线程
            long now = System.nanoTime();
            nanosTimeout -= now - lastTime; // 更新等待时间 循环遍历
            lastTime = now;
            if (Thread.interrupted())
                break;
        }
    } catch (RuntimeException ex) {
        cancelAcquire(node);
        throw ex;
    }
        // Arrive here only if interrupted
        cancelAcquire(node);
        throw new InterruptedException();
    }
```

这样通过 AQS 的协作，所有调用 future.get(long timeout, TimeUnit unit) 的线程都会按顺序等待，直到线成执行完被唤醒或者超时时间到 主动抛出异常。

总结

至此为止 FutureTask 的解析已经基本结束了，可以看到。它依靠 AQS 的共享锁实现了对线程执行结果的访问控制。和我们通常意义上的访问控制（并发访问某个资源，获取失败时，沉睡自己等待唤醒或者超时后返回）基本是一致的，不外乎维护了一个等待资源的列表。将等待资源的线程通过链表的方式串了起来。

当然 AQS 的功能远不仅如此，它还提供了一套独占锁的 API，帮助使用者实现独占锁的功能。

最常用的 Reentrantlock 就是使用这套 API 做的。

有机会的话再和大家分享下它的实现。

原文链接：http://www.liuinsect.com/2014/02/17/futuretask-

C++：在堆上创建对象，还是在栈上？

如果需要在堆上创建对象，要么使用 new 运算符，要么使用 malloc 系列函数。这点没有异议。

真正有异议的是下面的代码：

```
1                    Object obj;
```

此时，obj 是在栈上分配的吗？

要回答这个问题，我们首先要理解这个语句是什么意思。这个语句就是代表着，在栈上创建对象吗？

其实，这行语句的含义是，使对象 obj 具有“自动存储（automatic storage）”的性质。所谓“自动存储”，意思是这个对象的存储位置取决于其声明所在的上下文。

如果这个语句出现在函数内部，那么它就在栈上创建对象。

如果这个语句不是在函数内部，而是作为一个类的成员变量，则取决于这个类的对象是如何分配的。考虑下面的代码：

指针 pClass 所指向的对象在堆上分配空间。因为 Object obj;语句的含义是“自动存储”，所以，pClass->obj 也是在堆上创建的。

理解了这一点，再来看下面的语句：

```
1                    Object *pObj;
2                    pObj = new Object;
```

Object *pObj;代表，指针 pObj 是自动存储的，仅此而已，没有任何其它含义。而下面一行语句则指出，这个指针所指向的对象是在堆上面分配的。如果这两行语句出现在一个函数内部，意味着当函数结束时，pObj 会被销毁，但是它指向的对象不会。因此，为了继续使用这个对象，通常我们会在函数最后添加一个 return 语句，或者使用一个传出参数。否则的话，这个在堆上创建的对象就没有指针指向它，也就是说，这个对象造成了内存泄露。

并不是说指针指向的对象都是在堆上创建的。下面的代码则使用指针指向一个在栈上创建的对象

```
1                    Object obj;
```

```
2                                   Object *pObj = &obj;
```

至此，我们解释了函数内部的变量和成员变量。还有两类变量：全局变量和 static 变量。它们即不在堆上创建，也不在栈上创建。它们有自己的内存空间，是除堆和栈以外的数据区。也就是说，当 Object obj 即不在函数内部，又不是类的成员变量时，这个对象会在全局数据段创建，同理适用于 static 变量。对于指针 Object *pObj;，如果这个语句出现在函数内部或类的成员变量，正如我们前面所说的，这个指针是自动存储的。但是，如果这个语句是在类的外部，它就是在全局数据段创建的。虽然它指向的对象可能在堆上创建，也可能在栈上创建。

堆和栈的区别在于两点：

生命周期

第一点才是我们需要着重考虑的。由于栈的特性，如果你需要一个具有比其所在的上下文更长的生命周期的变量，只能在堆上创建它。所以，我们的推荐是：只要能在栈上创建对象，就在栈上创建；否则的话，如果你不得不需要更长的生命周期，只能选择堆上创建。这是由于在栈上的对象不需要我们手动管理内存。有经验的开发人员都会对内存管理感到头疼，我们就是要避免这种情况的发生。总的来说，我们更多推荐选择在栈上创建对象。

但是，有些情况，即便你在栈上创建了对象，它还是会占用堆的空间。考虑如下代码：

```
1                                   void func
2                                   {
3                                       std::vector v;
4                                   }
```

对象 v 是在栈上创建的。但是，STL 的 vector 类其实是在堆上面存储数据的（这点可以查看源代码）。因此，只有对象 v 本身是在栈上的，它所管理的数据（这些数据大多数时候都会远大于其本身的大小）还是保存在堆上。

关于第二点性能，有影响，不过一般可以忽略不计。确切的说，一般情况下你不需要考虑性能问题，除非它真的是一个问题。

首先，在堆上创建对象需要追踪内存的可用区域。这个算法是由操作系统提供，通常不会是常量时间的。当内存出现大量碎片，或者几乎用到 100% 内存

时，这个过程会变得更久。与此相比，栈分配是常量时间的。其次，栈的大小是固定的，并且远小于堆的大小。所以，如果你需要分配很大的对象，或者很多很多小对象，一般而言，堆是更好的选择。如果你分配的对象大小超出栈的大小，

通常会抛出一个异常。尽管很罕见，但是有时候也的确会发生。有关性能方面的问题，更多出现在嵌入式开发中：频繁地分配、释放内存可能造成碎片问题。

现代操作系统中，堆和栈都可以映射到虚拟内存中。在 32 位 Linux，我们可以把一个 2G 的数据放入堆中，而在 Mac OS 中，栈可能会限制为 65M。

总的来说，关于究竟在堆上，还是在栈上创建对象，首要考虑你所需要的生命周期。当性能真正成为瓶颈的时候，才去考虑性能的问题。堆和栈是提供给开发者的两个不同的工具，不存在一个放之四海而皆准的规则告诉你，一个对象必须放在堆中还是在栈中。选择权在开发者手中，决定权在开发者的经验中。

原文链接：http://www.devbean.net/2014/02/cpp-create-object-on-heap-or-stack/

[技术纵横]

UPYUN：用 Erlang 开发的对象存储系统

在国内的几家云计算创业公司当中，UPYUN（又拍云）选择了一个比较独特的定位：面向开发者提供非结构化数据云存储服务。非结构化数据存储服务一个很重要的卖点是要提供快速的静态文件访问能力，这对底层的存储系统性能和上层的 CDN 系统提出了较高的要求。

黄慧攀（@oneoo）是 UPYUN 技术总监。在 QCon 上海 2013 大会上，黄慧攀介绍了 UPYUN 的 CDN 系统架构，包括 Nginx 的二次开发经验、防盗链服务的实现、海量小文件的性能处理等；在 QCon 北京 2014 大会上，他将对 UPYUN 底层的对象存储系统的研发经验进行分享。

在本次采访中，黄慧攀介绍了 UPYUN 对象存储系统的一些历史，团队的分工，以及做测试方面的一些思路。

InfoQ：先介绍一下你自己吧。你关于计算机的知识都是自学，从底层网络、操作系统到上层的 Java、PHP 都玩，Lua 也玩。你对技术的选择有什么标准吗？如果有，是怎样的标准？

黄慧攀：我是出身于广东一个小城市“鹤山”人，最早是在 95 年接触电脑，98 年开始使用互联网，那时网易还只是做邮箱服务的，我非常感谢我的初中母校，使我能这么早期接触到互联网，影响一生 :) 也因为当年电脑、互联网才刚刚起步，学校也缺乏较好的教育能力，所以很多知识需要自学。也因为这个兴趣太浓，搞得其他学科基本都挂科了，也就没考上高中和大学。到现在还是有点小后悔，起码得把英文学好。

2001 年，18 岁的我第一份工作是市里一个集团公司的 B2B 门户网站，负责程序开发工作。那时用的语言是 PHP，边学边做的折腾了 3 年时间。

2004 年，项目因为市场、资金的原因结束了。在我们的小城市互联网就业机会基本为零，只好转到一个做弱电工程的公司任技术工程师，负责网络系统方案设计、智能灯光系统等等。

2006 年，压抑不住互联网的心，就出来创办 yo2.cn 优博网，国内第一个基于 WordPress 的博客服务平台，这个创业经历使我的技术能力提升很大，因为

没人嘛，所以整个网站的事情都得自己做，开发、运维、客服，甚至设计等等。记得当时网站被人吐槽最多的就是用户体验，我想如果能把这块也做好，可以做 UED 了，哈哈。

2009 年，机缘巧合来到杭州，跟朋友做了几次创业，虽然也是失败告终，但在其中的过程使自己成长了很多，因为创业嘛，所以很多事情都必须自己做的，这就奠定了我的技术层面比较广的基础。

2011 年，收到又拍网的邀请，开始又拍云的开发工作至今。

经历这么多年和多次创业，积累到比较丰富的技术经验。知识比较全面，在看待技术选型方面的把握还是比较准的。比如：Java 的优点是适合大型项目、团队协作开发，缺点也很明显，开发周期长、人员成本相对 PHP 高一些；而 PHP 的优点则是适合中小型项目、开发周期短、人员成本低，当然弊端也很明显，不支持多线程、系统资源占用高。每个语言都有自己的优缺点，要根据项目实际情况来选择。后来一个偶然的机会接触到 Lua 语言，发现它跟 PHP 很像，但又没了 PHP 几个大缺点，非常棒。所以现在我主要使用 C 和 Lua 这两个开发语言。

InfoQ：你自己做过博客平台，也在企业网站、网游等网站做过，现在在 UPYUN，可以说是从面向消费者的.com 公司转移到了一个更加基础一些的服务。你觉得在 UPYUN 做的事情跟以前有什么不一样吗？

黄慧攀：我觉得做 UPYUN 这件事，是之前几个项目的升华吧。因为这些面向消费者的项目让我知道在开发过程中产生的痛点，从而挖掘出开发者的需求。我很高兴能为开发者服务，帮助大家更快的把项目做好。

InfoQ：能不能简单介绍一下 UPYUN 这套对象存储系统的研发历程？比如是什么时候开始做的，最初的设计者是什么背景，借鉴过哪些思路，研发的过程中有没有什么好玩的故事等等。

黄慧攀：UPYUN 的对象存储系统其实早在 08 年就开始设计的，当初用的是 MogileFS，为又拍网服务。因为早期的 MogileFS 的设计本身有一定限制，tracker 角色的元信息使用单个 MySQL 实例存储，无法满足我们日益增长的存储量，所以在 2010 年转为使用 Erlang 语言开发。设计目标是提供 PB 级别的存储服务，经历 1 年多的业务测试才正式对外开放存储服务。

选择 Erlang 语言进行开发，主要是语言本身就支持分布式，这可以节省很多

开发工作。且 Erlang 语言在电信行业的应用非常广泛，稳定性有保障。

在分布式算法的选型上是参考 Dynamo 方案。而在具体的数据存储结构方面则是自主研发的一致性哈希算法，以实现多机柜、多服务器和多磁盘之间的数据备份工作。做到每文件的对应备份点在不同机柜、不同服务器上，避免某台服务器甚至某个机柜的服务器宕机而影响到文件的读写操作。

至于测试周期长达 1 年多，是因我们本身又拍网（照片社区）的数据量就非常庞大，从老的 MogileFS 集群迁移到新的云存储服务器占大部分时间，另外是因分布式存储服务的容灾测试过程比起应用测试要漫长得多，主要的测试点会有：某磁盘故障、某服务器故障、某机柜故障等好几种灾难测试，且每个故障都会产生一定量的数据迁移，文件会在集群内部自动寻找合适的备份点再建备份，所以说测试周期需要很长时间。也只有做到充分的测试，我们才放心的在集群上存储大量数据。否则等遇到无法排除的问题，要考虑新建集群的话，迁移成本和周期都会非常巨大。比如 10PB 的数据要从 A 集群迁移到 N 集群，网络传输就要 100pb，基于 10gb 网络也得耗时半年；且要保障迁移期间内不再发生新故障，这是很难做到的。所以我们选择前期测试做得非常充分，来保障日后服务的可持续性。

InfoQ：又拍云专注于做图片的存储，你们提供了一些很有特色的服务（如缩略图、防盗链），同时非常专注于服务质量。相对于文件备份类的应用场景，海量小图片存储是非常吃资源的，你们在存储系统的设计上做了哪些工作以确保在资源占用高的情况下仍然能保持图片访问的服务质量？

黄慧攀：是的，UPYUN 主要面向小于 100MB 的小文件提供服务，目前我们的存储集群已存有超过 2PB 的数据。面向海量小文件所面临的主要问题是：随机读取非常高、磁盘性能低；大家都知道缓存系统可以解决这类问题，而 CDN 其实就是个巨大的缓存系统，所以我们自建了 CDN 并对外提供服务。不仅能解决海量小文件所产生的磁盘性能问题，还能加速文件在互联网上的传输，一举两得。

InfoQ：UPYUN 系统的测试是如何做的？

黄慧攀：我们团队还比较小，目前未专门设立测试部门，所有测试工作均由项目开发者来完成，毕竟开发人员更清楚会有哪些潜在问题，并制定自动化测试的样例。下面是我们一个项目的开发、测试与发布流程：

项目策划、文档和方案撰写

开发（过程中会有两名以上开发人员交叉 review）

本地测试（主要测试该项目的功能是否正常和程序稳定性、资源占用率等等）

模拟平台测试（主要测试该项目的功能上线是否对原平台上其他子系统产生不良影响，这里会有我们自己编写的一批批量测试脚本，以验证平台每项功能逻辑是否正确）

灰度测试（业务环境中抽取 1%的服务器更新或指定某个别客户可使用该功能来进行测试）

全网发布

从整个流程来看，我们的测试周期是比较长的，测试工作占整个项目周期 50%以上，甚至个别影响范围大的项目，测试周期会长达半年以上。

InfoQ：你们的团队是怎样分工的？研发跟产品运营、系统运维的同学又是如何沟通的？

黄慧攀：大家从我们的产品介绍上会知道我们主要提供 3 块服务，

云存储

云分发

云处理

所以我们的开发团队主要是根据这 3 个方向进行分组。现在我们团队分应用开发组和核心研发组，而在核心研发组中又分存储、分发、处理 3 个小组，分得比较细。因此我们的小组成员之间会有交叉分工，以便大家对整体系统能有充分的了解。

我们的产品服务与一般互联网服务不太一样，我们是以产品为主导而非运营主导，且我们的产品经理也是开发出身的，所以在与开发团队的协作沟通上不会存在什么问题。另外的运维部门则是更加紧密，因我们正在开始整个平台的自动化运维系统开发，我们的开发人员已走到一线，跟运维人员一起探讨运维自动化系统的功能性问题，开发人员能亲身了解运维工作和痛点，并以此来驱动运维自动化系统的开发工作。

InfoQ：这次 QCon 北京，你希望面向哪些人群进行分享？他们能从你的分享中获得什么？

黄慧攀：很感谢 QCon 能让我们来继续跟大家做些云计算方面的分享。在上

一次的 QCon 上海大会我跟大家分享了又拍云的 CDN 技术，按我们公司的服务层次划分，这次的分享主题是我们在云存储系统的研发和构建过程中遇到的一些问题和经验。希望大家能通过我们这次的分享，对云存储能有更深入的了解，比如分布式算法、存储结构和日常维护等等。

原文链接：http://www.infoq.com/cn/news/2014/02/upyun-object-storage

百度面试

百度移动云可穿戴部门的面试经历，面试官都非常热情友好，一上来到弄的我挺不好意思的。下面记录一下自己的面试过程，因为我真的没啥面试经验，需要总结下。

1 面

Objective C runtime library：Objective C 的对象模型，block 的底层实现结构，消息发送，消息转发，这些都需要背后 C 一层的描述，内存管理。

Core Data：中多线程中处理大量数据同步时的操作。

Multithreading：什么时候处理多线程，几种方式，优缺点。

Delegate， Notification，KVO， other 优缺点

runtime 有一点追问，category，method 的实现机制，class 的载入过程。 1 面整体感觉不错，40 分钟不到，感觉回答的还可以。被通知一会儿二面。

2 面

二面的时间非常长，差不多将近 3 个小时，直接面到快下班了。1 面问的主要是知识点。2 面问主要考察的是设计解决方案的能力，另外辅助追问的方式，考察深度和广度，回答过程中需要列出适合的具体例子，方案还需要细致到具体

的关键的函数名称，方法。另外考察设计模式的理解，最后还考了算法。因为时间太长，这里记录一些重要的问题。

设计一个 progress bar 解决方案，追问到 Core Graphic、CGPath、maskLayer。

设计一个 popup view 被追问到 keyWindow、UIWindow 的 layer、UIView hierarchy。

从设计模式的角度分析 Delegate、Notification、KVO 的区别。被追问到自己写的 library 和开源的项目中用到哪些设计模式，为什么使用，有哪些好处和坏处，现在能否改进。

算是问题 3 的追问，设计一个方案来检测 KVO 的同步异步问题。willChange 和 didChange 的不同点，然后被追问到有没有其他地方也有类似情况，被追问到 Core Data 中 falut object。

这个是问题 4 的追问，设计一个 KVO 系统。

Multithreading，什么时候采用 Multithreading 方案，以及理由。追问到系统还有哪些在后台运行的 thread，被追问到 view life cycle、iOS6 之后的不同以及内存管理。

Multithreading 中常常遇到的问题，追问到死锁，优先级翻转，线程池等。

百度有一个亿级别的 APP 需要统计用户行为的日志系统。不使用数据库，只是使用普通文件，设计一个系统。被追问到内存映射文件。这个问题本来是服务器的问题，我表示从来没做过，回答很瞎。

算法考了 2 个。一个是如何求 2 个集合的交集。另一个是百亿数据中查找相同的数字以及出现的次数。

最后还补充了几个小问题

自己对可穿戴设备的感受

自己如果进入这个 team，自己准备做那方面的事情

为什么创业，自己未来规划

一会被告知 3 面，但是因为太晚，约到次日下午 3 面。

3 面

3 面的时间和 1 面差不多 40 分钟，问了几个问题，主要是考察精神层面的东西。

为什么做 Windows Mobile

为什么改 iOS

为什么来百度

为什么 iPhone 可以成功，那些吸引你

如何看待 AppStore 现在的生态圈

后面就是他说的多一些，介绍团队遇到的困难以及 14 年团队的打算。最后他给了我 2 句话的评价，我觉得还是蛮对的。

选择的时候都是经过深思熟虑的

有野心，wanna make a difference

有意思的是，他说他也 wanna make a difference。

总结

整体还是挺尴尬的，几乎所有的知识点都是 1，2 年前积累的，13 年积累的东西基本上没有，都是一些虚的东西。 2 面挺好的，暴露了自己不少问题，设计模式那部分几乎没有概念了。

最后

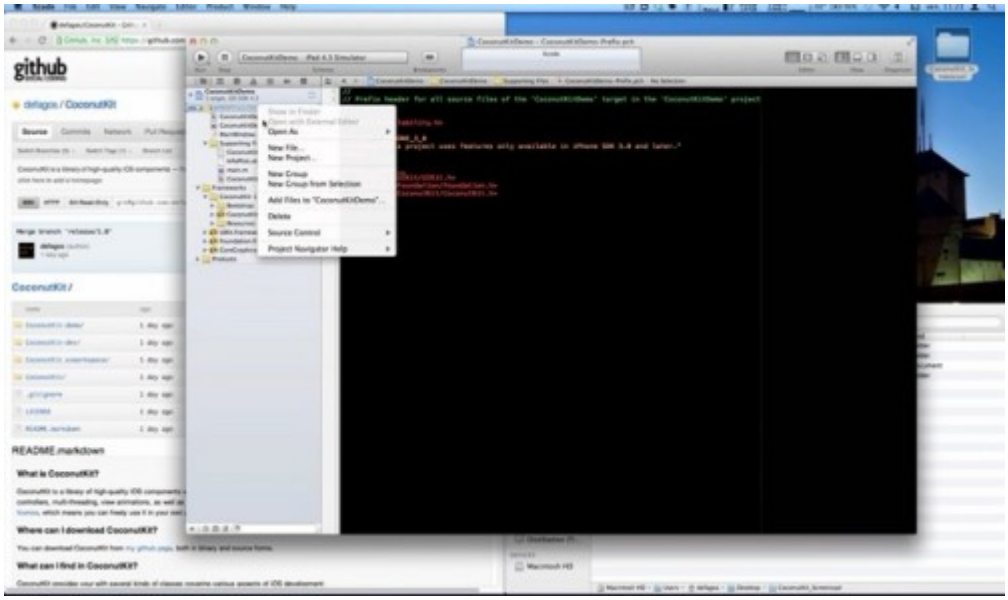
面试通过，我个人觉得 2 面我的问题在思考设计模式上面少，另外在 window hierarchy 上面有不足。设计 KVO 上面在 didChange 上面考虑不足。这些都是被当场戳穿的。 有一点疑虑的是整个面试中并没有问到 Core Animation。这个还是我蛮喜欢的部分。Anyway 个人很喜欢追问的方式，很容易考察出来理解的深度和广度。

整个面试收获很大，发现了很多不足。另外 1，2 面的面试题目个人觉得也不错。这里分享给大家。

原文链接：http://studentdeng.github.io/blog/2014/02/11/baidu-interview/

CoconutKit：iOS 开发必备的开源组件库

CoconutKit 是一款专门用于 iOS 开发的高质量开源组件库，基于 MIT 协议发布，代码业已托管到 GitHub 上。从简单的视图控制器到先进的本地化功能，CoconutKit 可以减少 iOS 开发者对样板代码（Boilerplate code）的编写，提高代码的质量和执行可靠的应用程序框架，这样，开发者就可以有更多的精力和时间来设计应用程序。



主要特性：

复杂的视图动画：用声明类的方式创建，由一些基于 UIView 块或基于 Core Animation 的子动画组成的动画。这些动画可以实现暂停、翻转、取消、重复等诸多功能。

高品质视图控制器容器：这些容器的功能甚至超过了 UIKit 内置容器。特别是，视图控制器可以组合或者堆叠，且可以使用任何种类的过渡动画。视图控制器中还包含 API，内容更加丰富，使用也很便捷。

Core Data 模型托管和验证更容易：只需通过引用上下文无关（context-free）文法作用于堆栈就可与托管对象上下文进行交互。Core Data 验证不再需要样板代码，且文本字段绑定表单的创建也不会很麻烦。

标签和按钮本土化，直接包含在 nib 文件中，无需创建和绑定。

原文链接：http://www.csdn.net/article/2014-02-20/2818483-iOS-components-CoconutKit

Associated Objects

Objective-C 开发者在遇到上面这条“咒语”相关的一些东西时，会不自觉的变的非常谨慎。一个主要原因是：弄乱 Objective-C 运行时可能会改变整个实现结构，因为所有的代码都是运行在它之上的。

一方面：<objc/runtime.h>中的函数可以给应用或者框架增加强大的新特性，这是通过其他方式不可能做到的。但另一方面：它会改变代码的正常运行逻辑和所有与之交互的东西（通常伴随着可怕的副作用）。

因而，这是我们认为进行这种魔鬼交易最大的恐惧点，下面来看一个 NSHipster 读者问得最多的一个主题：associated objects。

Associated Objects（关联对象）或者叫作关联引用（Associative References），是作为 Objective-C 2.0 运行时功能被引入到 Mac OS X 10.6 Snow Leopard（及 iOS4）系统。与它相关在<objc/runtime.h>中有 3 个 C 函数，它们可以让对象在运行时关联任何值：

objc_setAssociatedObject

objc_getAssociatedObject

objc_removeAssociatedObjects

为什么这几个方法很有用呢？因为开发者可以通过它们在分类中给已存在的类中添加自定义属性。

NSObject+AssociatedObject.h

```
@interface NSObject (AssociatedObject)
@property (nonatomic, strong) id associatedObject;
@end
```

NSObject+AssociatedObject.m

```
@implementation NSObject (AssociatedObject)
@dynamic associatedObject;
- (void)setAssociatedObject:(id)object {
        objc_setAssociatedObject(self, @selector(associatedObject), object,
```

OBJC_ASSOCIATION_RETAIN_NONATOMIC);

}

- (id)associatedObject {

return objc_getAssociatedObject(self, @selector(associatedObject));

}

通常推荐 key 使用 static char 类型——使用指针或许更好，key 值是一个唯一的常量，并只在 getters 和 setters 方法内部使用：

static char kAssociatedObjectKey;

objc_getAssociatedObject(self, &kAssociatedObjectKey);

然而，一个更简单的方案是：直接使用选择器（selector）。

因为 SEL 生成的时候就是一个唯一的常量，你可以使用 _cmd 作为 objc_setAssociatedObject()的 key。

—— Bill Bumgarner(@bbum) August28, 2009

关联对象的特性

被关联到对象的值根据使用的 objc_AssociationPolicy 类型不同表现出不同的特性：

Behavior	对应的@property 类型	描述
OBJC_ASSOCIATION_ASSIGN	@property (assign) 或 @property(unsafe_unretained)	给关联对象指定若引用
OBJC_ASSOCIATION_RETAIN_NONATOMIC	@property (nonatomic, strong)	给关联对象指定非原

		子的强引用
OBJC_ASSOCIATION_COPY_NONATOMIC	@property (nonatomic, copy)	给关联对象指定非原子的 copy 特性
OBJC_ASSOCIATION_RETAIN	@property (atomic, strong)	给关联对象指定原子的强引用
OBJC_ASSOCIATION_COPY	@property (atomic, copy)	给关联对象指定原子 copy 特性

通过 OBJC_ASSOCIATION_ASSIGN 分配的弱关联对象并不是完全和 weak 修饰符引用一样（对象初始化与释放时被置空），反而更像是 unsafe_unretained，所以你需要在访问弱关联对象时稍微注意一下。

根据 WWDC2011,Session322 对对象释放时间的描述，associated objects 清除

在对象生命周期中很晚才执行，通过被 NSObject -dealloc 方法调用的 object_dispose()函数完成。

移除关联对象

一个的方法是试图在某个时刻调用 objc_removeAssociatedObjects()函数来移除关联对象，然而，根据苹果文档描述，你不大可能有需求要自己去调用：

这个函数的主要目的是很容易的让对象恢复成它“原始状态”，你不应该使用它来移除关联的对象，因为它也会移除掉包括其他地方加入的全部的关联对象。所以一般你只需要通过调用 objc_setAssociatedObject 并传入 nil 值来清除关联值。

模式

添加私有变量来帮助实现细节 。当拓展一个内置类时，可能有必要跟踪一些额外的状态，这是关联对象最普遍的应用场景。例如：AFNetworking 中在 UIImageView 的分类中使用关联对象来存储一个请求操作对象（operation object），用于异步的从远程获取图片。

添加公共属性来设置分类的特性 。有时候，通过添加一个属性让一个分类更加灵活，而不是作为函数参数。这种情况下，使用关联对象作为一个公开的属性是可接受的解决方案。还是拿前面 AFNetworking 的例子来说，UIImageView 的分类中 imageResponseSerializer 属性允许图片视图随意的使用一个过滤器，或者在图片请求并缓存之前就可以修改它的渲染。

为 KVO 创建一个关联的观察者（observer）。当在一个分类中使用 KVO 的时候，推荐使用一个自定义的关联对象作为观察者，而不是对象自己观察自己。

反模式

在不必要的时候使用关联对象。使用视图时一个常见的情况是通过数据模型或一些复合的值来创建一个便利的方法设置填充字段或属性。如果这些值在后面不会再被使用到，最好就不要使用关联对象了。

使用关联对象来保存一个可以被推算出来的值。例如，有人可能想通过关联对象存储 UITableViewCell 上一个自定义 accessoryView 的引用，使用 tableView:accessoryButtonTappedForRowWithIndexPath: 和 cellForRowAtIndexPath:即可以达到要求。

使用关联对象来代替 X。其中 X 代表下面的一些项：

子类化，当使用继承比使用组合更合适的时候。

Target-Action 给响应者添加交互事件。

手势识别，当 target-action 模式不够用的时候。

代理，当事件可以委托给其他对象。

消息 & 消息中心使用低耦合的方式来广播消息。

关联对象应该被当做最后的手段来使用（不得不用时才用），而不是为了寻求一个解决方案就行（事实上，分类本身就不应该是解决问题优先选择的工具）。

像一些巧妙的伎俩、hack 手段或者是变通的解决方案，人们总是倾向于创造机会来使用他们——特别是刚刚接触他们时。尽可能的在理解并领悟之后再做出正确的方案，避免自己陷入一知半解的尴尬处境。

原文链接：http://esoftmobile.com/2014/02/18/associated-objects/

[技术翻译]构建现代化的 Objective-C (上)

当学习一个新技能时，比如编程语言，我们经常为了能运行，而把所有能用的都揉合在一起。再后来，我们回归到这些习惯，并进行重新估计，采用社区中的最佳实践并写出更好、更有结构化的代码。

最近，Objective-C 语言收到了过多的新特性，但社区的最佳实践还没持续更新。这就超出了“风格”的范畴，进入了“结构”的领域。

我在最近一段时间里，审视了我自己的代码实践，评估了我可以在哪里做的更好，所以我想我应该和你们分享我的发现。

欢迎光临现代 Objective-C。

（以上部分感谢 Ley 翻译）

访问实例变量(Instance Variable)

唉，实例变量。从何说起呢。一句话，实例变量很糟。如果你会这样写：

```
@interface MyClass : NSObject {
    BOOL someVariable;
}
@end
```

别这么写了。现在就改。

不要再声明实例变量了，尤其别声明在头文件里。你应该把它们声明为 property，然后用消息机制或者“.”来访问它们。

我之前说过为什么通过实例变量来访问 property 没有明显的益处。事实上，通过 getter/setter 方法来访问有几个优势。

一致性：你不需要再去怀疑 getter/setter 方法有没有副作用了，假如有的话，也肯定早就会被人注意到。

Debug：你可以简便地在 getter/setter 方法上设置断点，而不用在运行时针对实例变量的内存地址设置 watchpoint。

真的，没有理由再去声明实例变量、再去直接访问这些实例变量形式的属性了——除非是在本身覆盖(override) getter/setter 的方法里，或者在 initializer/dealloc 方法里，取决于你想要代码有多少防御性(感谢 Bryan 提供链接)。再除非就是习惯了，但你应该改掉这个习惯。我就改了。

更新：我找到了官方文档的一个链接，建议不要在 dealloc 中调用 getter/setter 方法。供参考。

那么只读的属性怎么办呢？既然没有 setter 方法，你不还是需要访问实例变量吗？好问题。这引出了本文的下一点。

在头文件中定义 readonly 属性

在你 public 的接口中要暴露属性或组件，使用 readonly 属性是一个很好的方式。但是如果不直接访问实例变量，怎么设置它们的值呢？答案是在 .m 文件中定义一个 private 的 class extension。

在你的头文件中，声明如下：

@interface MyClass : NSObject

@property (nonatomic, readonly) Type propertyName;

@end

然后，在类的 implementation 文件中，在以上定义的基础上，再定义如下：

@interface MyClass ()

// Private Access

@property (nonatomic, strong, readwrite) Type propertyName;

@end

这样你就定义了 public 的 getter 和 private 的 setter。可喜可贺！现在你不需要再访问实例变量了。

Schwa 补充了一条建议:

@ashfurrow 回复:《在头文件中定义 readonly 属性》。我还要加一条，不要在头文件里暴露可变(mutable)对象。顺便赞一下，很好的文章。

— Jonathan Wight (@schwa) January 24, 2014

合理地定义 BOOL 类型的属性

定义属性时，遵从 Apple 官方指南总是没错的。但我承认，我也不总是看得那么勤快。要记住，定义 BOOL 类型的属性时，要同时手动定义一个 getter 方法。

@property (nonatomic, assign, getter = isSomething) BOOL something;

如非必要，不要把#import 写在头文件里

我经常在 Objective-C 新手写的代码里看到这种情况。总体来说，问题在于大多数的#import 语句应该只写在 .m 文件里，而不应该写在 .h 头文件里。

如下面的例子。

#import "MyOtherClass.h"

@interface MyClass : NSObject

@property (nonatomic, strong) MyOtherClass property;

@end

你可以改写代码如下，然后在类的 implementation 文件 yourMyClass.m 中再去 #import MyOtherClass.h 头文件。

@class MyOtherClass;

@interface MyClass : NSObject

@property (nonatomic, strong) MyOtherClass property;

@end

@class MyOtherClass 的写法是类的前向声明(forward class declaration)。

这样写的益处良多。用类的前向声明来代替 #import 头文件可以提高编译速度，可以避免循环#import，还可以让你的头文件更轻盈——本该如此。

原文链接：http://www.cnblogs.com/hamhog/p/3561514.html

iOS 的后台运行和多任务处理

好久之前整理了一篇 iOS 运行状态的文章，其中略微提到了应用程序在后台时的情况。本文将本人对 iOS 的后台运行和多任务处理支持的个人理解和实践做一个稍微详细一点的整理。网上关于 iOS 后台运行和多任务支持的文章不少，但其中很多都是转来转去甚至是抄来抄去，本篇虽不能面面俱到，但可以说是本人结合苹果官方的几篇文档理解后的一个原创。

0. iOS 对后台支持的历史背景

根据苹果目前的文档来看，大致可以将 iOS 从最开始到 iOS7.0 后的版本对后台任务支持分为三大阶段，分别是：

iOS4.0 以前。据说这个阶段 iOS 是完全没有后台的概念的，只有一个不受前后台影响的推送功能，只要在 iPhone 上按下了圆圆的 Home 键，应用直接被关

掉。这个阶段我只能是根据老苹果用户的文章来推断了，因为我本人近距离接触 iOS 也是在 4.0 之后的时候。

iOS4.0 以后，7.0 之前。苹果现存的后天运行和多任务支持的基本上都是 4.0 开始的，可以说这是一个重要的分界点。这一阶段的特点是应用有后台的概念，按 Home 键之后应用不退出，但冻结，网络上有一个名称叫做“墓碑式”

iOS7.0 之后。新增了机种后台运行模式，智能调度。

既然现在大多数的情况都是 4.0 之后的特征，下文我们就详细来看。

1. “禳星续命”

《三国演义》第一百零三章标题为“上方谷司马受困，五丈原诸葛禳星”，对三国有点了解的人都知道诸葛亮禳星续命的经典故事。大意是讲诸葛亮在五丈原和敌军僵持，但知道自己命不久矣，为了“酬三顾”希望能够让自己再多活些时日以胜此仗，至于结局这里就不用说了。

那么当 iOS(4.0 之后)的应用状态将要从 Active 经由 Inactive 退到 Background 的时候，能不能争取到一些执行时间来完成“墓碑式”冻结前最后的任务呢？苹果给了这个机会。

在前面一篇文章“iOS 应用程序的状态及其切换（生命周期）”中整理了生命周期相关的方法，在程序进入后台的时候，方法：

```
1    - (void)applicationDidEnterBackground:(UIApplication *)application
```

会被调用从而开始执行后台运行的程序，官方文档中要求此方法需在 5 秒内结束。

按文档给出的示例，可以给出如下调用：

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    bgTask = [application beginBackgroundTaskWithExpirationHandler:^{
        // Clean up any unfinished task business by marking where you
        // stopped or ending the task outright.
        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    }];

    // Start the long-running task and return immediately.

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{

        // Do the work associated with the task, preferably in chunks.

        [application endBackgroundTask:bgTask];
        bgTask = UIBackgroundTaskInvalid;
    });
}
```

这种方式是苹果推荐的方式，比较优雅。我个人用 iOS7 的真机调试，默认情况下会在后台跑 3 分钟，然后停掉。

大家如果感兴趣也可以尝试在 applicationDidEnterBackground 方法中使用主线程跑一个足够长的任务，看看会怎样。

2. 真后台和服务

iOS 从诞生到现在对应用的后台运行一直有所限制，了解苹果产品的朋友应该都明白，这并不是技术上的问题，而正是苹果为了提高用户体验而做的取舍，为了手机的资源更合理的利用，而不是被后台运行的“卡死”或者手机电量无辜消耗。

既然传说中 4.0 之前的时代都有推送，那么 4.0 之后肯定也是要支持的。消息推送可是把客户端和服务器紧密结合起来满足商业需求的重要利器。除了消息推送通知，iOS 也有对应的本地通知。

实际上，除了消息意外，苹果的 iOS 对特定的服务也是支持真后台运行的。主要包括：

音乐播放和录制，比如你打开虾米播放器再退到后台，音乐也还是播放着的

定位服务，iOS 的定位服务分为显著位置变化检测和高精确度的定位，后者是支持后台定位跟踪的

VoIP

当然，真后台的服务也不止这些，这些只是比较常见的而已，更详细的可以参看这里：

App States and Multitasking

3. iOS7.0 的新东西

实际上本人对 iOS7 中后台运行的新特点并没有太多应用实践经验，大家所瞩目的主要是叫做 Background Fetch 的服务。iOS 操作系统可以根据应用在后台的运行情况智能调度任务，并进行反馈统计。

原文链接：http://www.molotang.com/articles/1480.html

AFNetworking 2.0 Tutorial

In iOS 7, Apple introduced NSURLSession as the new, preferred method of networking (as opposed to the older NSURLConnection API). Using this raw NSURLSession API is definitely a valid way to write your networking code – we even have a tutorial on that.

However, there's an alternative to consider – using the popular third party networking library AFNetworking.

The latest version of AFNetworking (2.0) is now built on top of NSURLSession, so you get all of the great features provided there. But you also get a lot of extra cool features – like serialization, reachability support, UIKit integration (such as a handy category on asynchronously loading images in a UIImageView), and more.

AFNetworking is incredibly popular – it won our Reader's Choice 2012 Best iOS Library Award. It's also one of the most widely used, open-source projects with over 10,000 stars, 2,600 forks, and 160 contributors on Github.

In this AFNetworking 2.0 tutorial, you will learn about the major components of AFNetworking by building a Weather App that uses feeds from World Weather Online. You'll start with static weather data, but by the end of the tutorial, the app will be fully connected to live weather feeds.

Today's forecast: a cool developer learns all about AFNetworking and gets inspired to use it in his/her apps. Let's get busy!

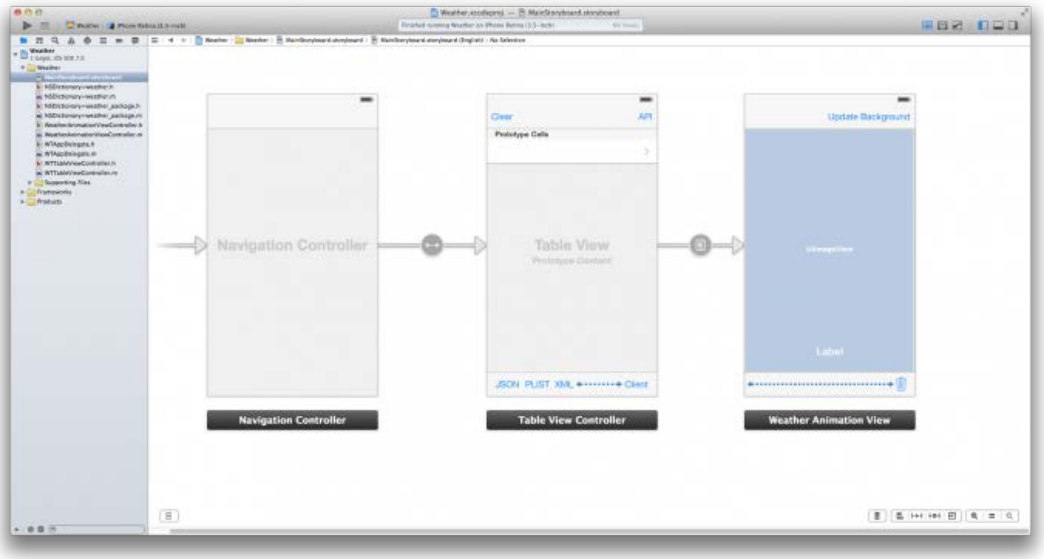
Getting Started

First download the starter project for this AFNetworking 2.0 tutorial here.

This project provides a basic UI to get you started – no AFNetworking code has been added yet.

Open MainStoryboard.storyboard, and you will see three view

controllers:



From left to right, they are:

A top-level navigation controller

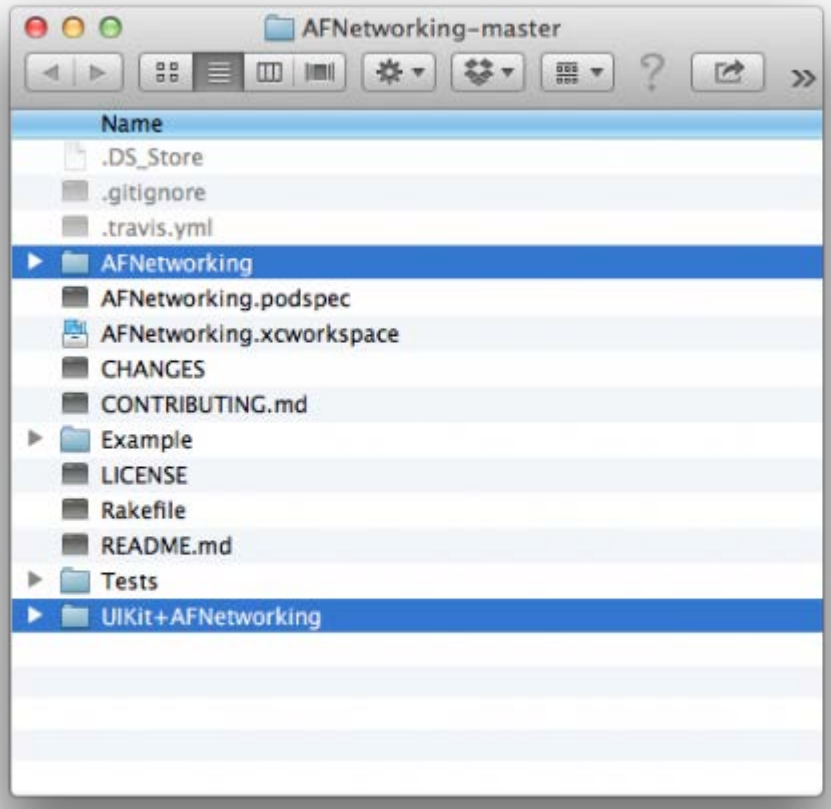
A table view controller that will display the weather, one row per day

A custom view controller (WeatherAnimationViewController) that will show the weather for a single day when the user taps on a table view cell

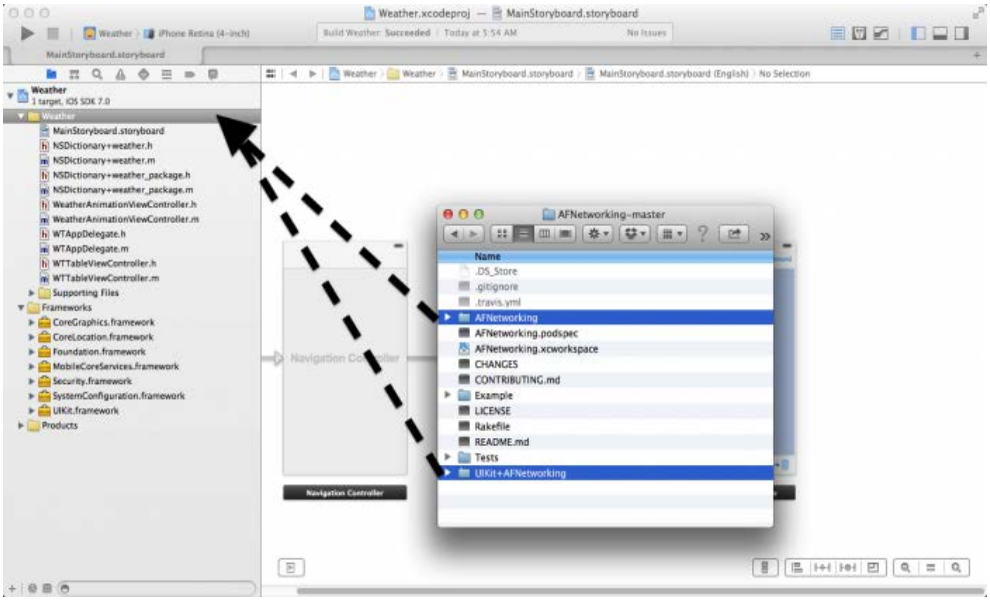
Build and run the project. You'll see the UI appear, but nothing works yet. That's because the app needs to get its data from the network, but this code hasn't been added yet. This is what you will be doing in this tutorial!

The first thing you need to do is include the AFNetworking framework in your project. Download the latest versionfrom GitHub by clicking on the Download Zip link.

When you unzip the file, you will see that it includes several subfolders and items. Of particular interest, it includes a subfolder called AFNetworking and another called UIKit+AFNetworking as shown below:



Drag these folders into your Xcode project.



When presented with options for adding the folders, make sure that Copy items into destination group's folder (if needed) and Create groups

for any added folders are both checked.

To complete the setup, open the pre-compiled header Weather-Prefix.pch from the Supporting Files section of the project. Add this line after the other imports:

```
#import "AFNetworking.h"
```

Adding AFNetworking to the pre-compiled header means that the framework will be automatically included in all the project's source files.

Pretty easy, eh? Now you're ready to "weather" the code!

Operation JSON

AFNetworking is smart enough to load and process structured data over the network, as well as plain old HTTP requests. In particular, it supports JSON, XML and Property Lists (plists).

You could download some JSON and then run it through a parser (like the built-in NSJSONSerialization) yourself, but why bother? AFNetworking can do it all!

First you need the base URL of the test script. Add this to the top of WTTableViewController.m, just underneath all the #import lines.

```
static NSString * const BaseURLString = @"http://www.raywenderlich.com/demos/weather_sample/";
```

This is the URL to an incredibly simple "web service" that I created for you for this tutorial. If you're curious what it looks like, you can download the source.

The web service returns weather data in three different formats - JSON, XML, and PLIST. You can take a look at the data it can return by using these URLS:

http://www.raywenderlich.com/demos/weather_sample/weather.php?format=json

http://www.raywenderlich.com/demos/weather_sample/weather.php?for

mat=xml

http://www.raywenderlich.com/demos/weather_sample/weather.php?format=plist (might not show correctly in your browser)

The first data format you will be using is JSON. JSON is a very common JavaScript-derived object format. It looks something like this:

```
{
    "data": {
        "current_condition": [
            {
                "cloudcover": "16",
                "humidity": "59",
                "observation_time": "09:09 PM",
            }
        ]
    }
}
```

Note: If you'd like to learn more about JSON, check out our Working with JSON Tutorial.

When the user taps the JSON button, the app will load and process JSON data from the server. InWTTableViewController.m, find the jsonTapped: method (it should be empty) and replace it with the following:

```
- (IBAction)jsonTapped:(id)sender
{
    // 1
    NSString *string = [NSString stringWithFormat:@"%@weather.php?format=json", BaseURLString];
    NSURL *url = [NSURL URLWithString:string];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
```

```
        // 2
        AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation
alloc] initWithRequest:request];
        operation.responseSerializer = [AFJSONResponseSerializer
serializer];
        [operation
setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *operation, id
responseObject) {
            // 3
            self.weather = (NSDictionary *)responseObject;
            self.title = @"JSON Retrieved";
            [self.tableView reloadData];
        } failure:^(AFHTTPRequestOperation *operation, NSError *error)
{
            // 4
            UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error Retrieving Weather"
message:[error localizedDescription]
delegate:nil
cancelButtonTitle:@"Ok"
otherButtonTitles:nil];
            [alertView show];
        }];
        // 5
        [operation start];
    }
```

Awesome, this is your first AFNetworking code! Since this is all new, I'll explain it one section at a time.

You first create a string representing the full url from the base URL

string. This is then used to create an NSURL object, which is used to make an NSURLRequest.

AFHTTPRequestOperation is an all-in-one class for handling HTTP transfers across the network. You tell it that the response should be read as JSON by setting the responseSerializer property to the default JSON serializer. AFNetworking will then take care of parsing the JSON for you.

The success block runs when (surprise!) the request succeeds. The JSON serializer parses the received data and returns a dictionary in the responseObject variable, which is stored in the weather property.

The failure block runs if something goes wrong - such as if networking isn't available. If this happens, you simply display an alert with the error message.

You must explicitly tell the operation to "start" (or else nothing will happen).

As you can see, AFNetworking is extremely simple to use. In just a few lines of code, you were able to create a networking operation that both downloads and parses its response.

Now that the weather data is stored in self.weather, you need to display it. Find thetableView:numberOfRowsInSection: method and replace it with the following:

```
    - (NSInteger)tableView:(UITableView *)tableView
numberOfRowsInSection:(NSInteger)section
    {
        if(!self.weather)
            return 0;
        switch (section) {
            case 0: {
                return 1;
            }
```

```
            case 1: {
                NSArray *upcomingWeather = [self.weather
upcomingWeather];
                return [upcomingWeather count];
            }
            default:
                return 0;
        }
    }
```

The table view will have two sections: the first to display the current weather and the second to display the upcoming weather.

"Wait a minute!", you might be thinking. What is this [self.weather upcomingWeather]? If self.weather is a plain old NSDictionary, how does it know what "upcomingWeather" is?

To make it easier to display the data, I added a couple of helper categories on NSDictionary in the starter project:

NSDictionary+weather

NSDictionary+weather_package

These categories add some handy methods that make it a little easier to access the data elements. You want to focus on the networking part and not on navigating NSDictionary keys, right?

Note: FYI, an alternative way to make working with JSON results a bit easier than looking up keys in dictionaries or creating special categories like this is to use a third party library like JSONModel.

Still in WTTableViewController.m, find the tableView:cellForRowAtIndexPath: method and replace it with the following implementation:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath
```

```
    {
        static NSString *CellIdentifier = @"WeatherCell";
        UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:CellIdentifier
forIndexPath:indexPath];
        NSDictionary *daysWeather = nil;
        switch (indexPath.section) {
            case 0: {
                daysWeather = [self.weather currentCondition];
                break;
            }
            case 1: {
                NSArray *upcomingWeather = [self.weather
upcomingWeather];
                daysWeather = upcomingWeather[indexPath.row];
                break;
            }
            default:
                break;
        }
        cell.textLabel.text = [daysWeather weatherDescription];
        // You will add code here later to customize the cell, but it's
good for now.
        return cell;
    }
```

Like the tableView:numberOfRowsInSection: method, the handy NSDictionary categories are used to easily access the data. The current day's weather is a dictionary, and the upcoming days are stored in an array.

Build and run your project; tap on the JSON button to get the networking request in motion; and you should see this:

JSON success!

Operation Property Lists

Property lists (or plists for short) are just XML files structured in a certain way (defined by Apple). Apple uses them all over the place for things like storing user settings. They look something like this:

```
<dict>
  <key>data</key>
  <dict>
    <key>current_condition</key>
```

```
        <array>
        <dict>
          <key>cloudcover</key>
          <string>16</string>
          <key>humidity</key>
          <string>59</string>
          ...
```

The above represents:

A dictionary with a single key called “data” that contains another dictionary.

That dictionary has a single key called “current_condition” that contains an array.

That array contains a dictionary with several keys and values, like cloudcover=16 and humidity=59.

It’s time to load the plist version of the weather data. Find the plistTapped: method and replace the empty implementation with the following:

```
- (IBAction)plistTapped:(id)sender
{
    NSString *string = [NSString
stringWithFormat:@"%@weather.php?format=plist", BaseURLString];
    NSURL *url = [NSURL URLWithString:string];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];
    AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation
alloc] initWithRequest:request];
    // Make sure to set the responseSerializer correctly
    operation.responseSerializer =
[AFPropertyListResponseSerializer serializer];
```

```
        [operation
setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *operation, id
responseObject) {
            self.weather = (NSDictionary *)responseObject;
            self.title = @"PLIST Retrieved";
            [self.tableView reloadData];
        } failure:^(AFHTTPRequestOperation *operation, NSError *error)
{
            UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error Retrieving Weather"
message:[error localizedDescription]
delegate:nil
cancelButtonTitle:@"Ok"
otherButtonTitles:nil];
            [alertView show];
        }];
        [operation start];
    }
```

Notice that this code is almost identical to the JSON version, except for changing the responseSerializer to the default AFPropertyListResponseSerializer to let AFNetworking know that you’re going to be parsing a plist.

That’s pretty neat: your app can accept either JSON or plist formats with just a tiny change to the code!

Build and run your project and try tapping on the PLIST button. You should see something like this:

The Clear button in the top navigation bar will clear the title and table view data so you can reset everything to make sure the requests are going through.

Operation XML

While AFNetworking handles JSON and plist parsing for you, working with XML is a little more complicated. This time, it's your job to construct the weather dictionary from the XML feed.

Fortunately, iOS provides some help via the NSXMLParser class (which is a SAX parser, if you want to read up on it).

Still in WTTableViewController.m, find the xmlTapped: method and replace its implementation with the following:

```
- (IBAction)xmlTapped:(id)sender
```

```
    {
        NSString *string = [NSString
stringWithFormat:@"%@weather.php?format=xml", BaseURLString];
        NSURL *url = [NSURL URLWithString:string];
        NSURLRequest *request = [NSURLRequest requestWithURL:url];
        AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation
alloc] initWithRequest:request];
        // Make sure to set the responseSerializer correctly
        operation.responseSerializer = [AFXMLParserResponseSerializer
serializer];
        [operation
setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *operation, id
responseObject) {
            NSXMLParser *XMLParser = (NSXMLParser *)responseObject;
            [XMLParser setShouldProcessNamespaces:YES];
            // Leave these commented for now (you first need to add the
delegate methods)
            // XMLParser.delegate = self;
            // [XMLParser parse]
        } failure:^(AFHTTPRequestOperation *operation, NSError *error)
{
            UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error Retrieving Weather"
message:[error localizedDescription]
delegate:nil
cancelButtonTitle:@"Ok"
otherButtonTitles:nil];
            [alertView show];
```

```
        }];
        [operation start];
    }
```

This should look pretty familiar by now. The biggest change is that in the success block you don't get a nice, preprocessed NSDictionary object passed to you. Instead, responseObject is an instance of NSXMLParser, which you will use to do the heavy lifting in parsing the XML.

You'll need to implement a set of delegate methods for NXMLParser to be able to parse the XML. Notice that XMLParser's delegate is set to self, so you will need to add NSXMLParser's delegate methods to WTTableViewController to handle the parsing.

First, update WTTableViewController.h and change the class declaration at the top as follows:

```
    @interface WTTableViewController :
UITableViewController<NSXMLParserDelegate>
```

This means the class will implement the NSXMLParserDelegate protocol. You will implement these methods soon, but first you need to add a few properties.

Add the following properties to WTTableViewController.m within the class extension, right after @interface WTTableViewController () :

```
    @property(nonatomic, strong) NSMutableDictionary
*currentDictionary;   // current section being parsed
    @property(nonatomic, strong) NSMutableDictionary *xmlWeather;
// completed parsed xml response
    @property(nonatomic, strong) NSString *elementName;
    @property(nonatomic, strong) NSMutableString *outstring;
```

These properties will come in handy when you're parsing the XML.

Now paste this method in WTTableViewController.m, right before @end:

```
- (void)parserDidStartDocument:(NSXMLParser *)parser
{
    self.xmlWeather = [NSMutableDictionary dictionary];
}
```

The parser calls this method when it first starts parsing. When this happens, you set self.xmlWeather to a new dictionary, which will hold hold the XML data.

Next paste this method right after this previous one:

```
- (void)parser:(NSXMLParser *)parser didStartElement:(NSString
*)elementName namespaceURI:(NSString *)namespaceURI
qualifiedName:(NSString *)qName attributes:(NSDictionary
*)attributeDict
{
    self.elementName = qName;
    if([qName isEqualToString:@"current_condition"] ||
       [qName isEqualToString:@"weather"] ||
       [qName isEqualToString:@"request"]) {
        self.currentDictionary = [NSMutableDictionary
dictionary];
    }
    self.outstring = [NSMutableString string];
}
```

The parser calls this method when it finds a new element start tag. When this happens, you keep track of the new element's name as self.elementName and then set self.currentDictionary to a new dictionary if the element name represents the start of a new weather forecast. You also reset outstring as a new mutable string in preparation for new XML to be received related to the element.

Next paste this method just after the previous one:

```
- (void)parser:(NSXMLParser *)parser foundCharacters:(NSString
*)string
{
    if (!self.elementName)
        return;
    [self.outstring appendFormat:@"%@", string];
}
```

As the name suggests, the parser calls this method when it finds new characters on an XML element. You append the new characters to outstring, so they can be processed once the XML tag is closed.

Again, paste this next method just after the previous one:

```
- (void)parser:(NSXMLParser *)parser didEndElement:(NSString
*)elementName namespaceURI:(NSString *)namespaceURI
qualifiedName:(NSString *)qName
{
    // 1
    if ([qName isEqualToString:@"current_condition"] ||
        [qName isEqualToString:@"request"]) {
        self.xmlWeather[qName] = @[self.currentDictionary];
        self.currentDictionary = nil;
    }
    // 2
    else if ([qName isEqualToString:@"weather"]) {
        // Initialize the list of weather items if it doesn't
exist
        NSMutableArray *array = self.xmlWeather[@"weather"] ?:
[NSMutableArray array];
        // Add the current weather object
        [array addObject:self.currentDictionary];
```

```
            // Set the new array to the "weather" key on xmlWeather
dictionary
            self.xmlWeather[@"weather"] = array;
            self.currentDictionary = nil;
        }
        // 3
        else if ([qName isEqualToString:@"value"]) {
            // Ignore value tags, they only appear in the two
conditions below
        }
        // 4
        else if ([qName isEqualToString:@"weatherDesc"] ||
                 [qName isEqualToString:@"weatherIconUrl"]) {
            NSDictionary *dictionary = @{@"value": self.outstring};
            NSArray *array = @[dictionary];
            self.currentDictionary[qName] = array;
        }
        // 5
        else if (qName) {
            self.currentDictionary[qName] = self.outstring;
        }
        self.elementName = nil;
    }
```

This method is called when an end element tag is encountered. When that happens, you check for a few special tags:

The current_condition element indicates you have the weather for the current day. You add this directly to the xmlWeather dictionary.

The weather element means you have the weather for a subsequent day. While there is only one current day, there may be several subsequent days,

so you add this weather information to an array.

The value tag only appears inside other tags, so it's safe to skip over it.

The weatherDesc and weatherIconUrl element values need to be boxed inside an array before they can be stored. This way, they will match how the JSON and plist versions of the data are structured exactly.

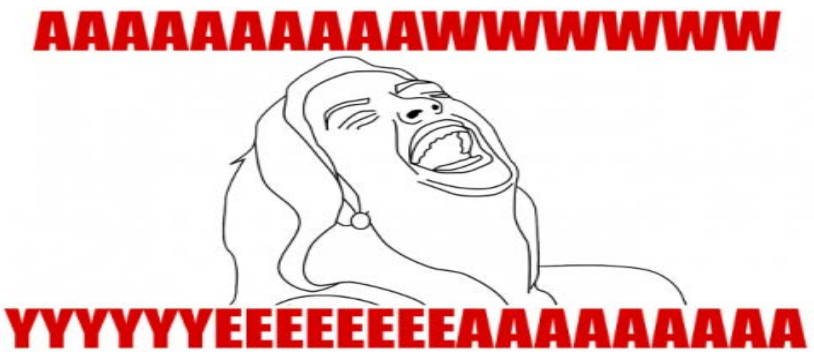
All other elements can be stored as is.

Now for the final delegate method! Paste this method just after the previous one:

```
- (void) parserDidEndDocument:(NSXMLParser *)parser
{
    self.weather = @{@"data": self.xmlWeather};
    self.title = @"XML Retrieved";
    [self.tableView reloadData];
}
```

The parser calls this method when it reaches the end of the document. At this point, the xmlWeather dictionary that you've been building is complete, so the table view can be reloaded.

Wrapping xmlWeather inside another NSDictionary might seem redundant, but this ensures the format matches up exactly with the JSON and plist versions. This way, all three data formats can be displayed with the same code!



Now that the delegate methods and properties are in place, return to the xmlTapped: method and uncomment the lines of code from before:

```
- (IBAction)xmlTapped:(id)sender
{
    ...
    [operation
setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *operation, id
responseObject) {
        NSXMLParser *XMLParser = (NSXMLParser *)responseObject;
        [XMLParser setShouldProcessNamespaces:YES];
        // These lines below were previously commented
        XMLParser.delegate = self;
        [XMLParser parse];
    ...
}
```

Build and run your project. Try tapping the XML button, and you should see this:

A Little Weather Flair

Hmm, that looks dreary, like a week's worth of rainy days. How could you jazz up the weather information in your table view?

Take another peak at the JSON format from before, and you will see that there are image URLs for each weather item. Displaying these weather images in each table view cell would add some visual interest to the app.

AFNetworking adds a category to UIImageView that lets you load images asynchronously, meaning the UI will remain responsive while images are downloaded in the background. To take advantage of this, first add the category import to the top of WTTableViewController.m:

```
#import "UIImageView+AFNetworking.h"
```

Find the tableView:cellForRowAtIndexPath: method and paste the

following code just above the final return cell; line (there should be a comment marking the spot):

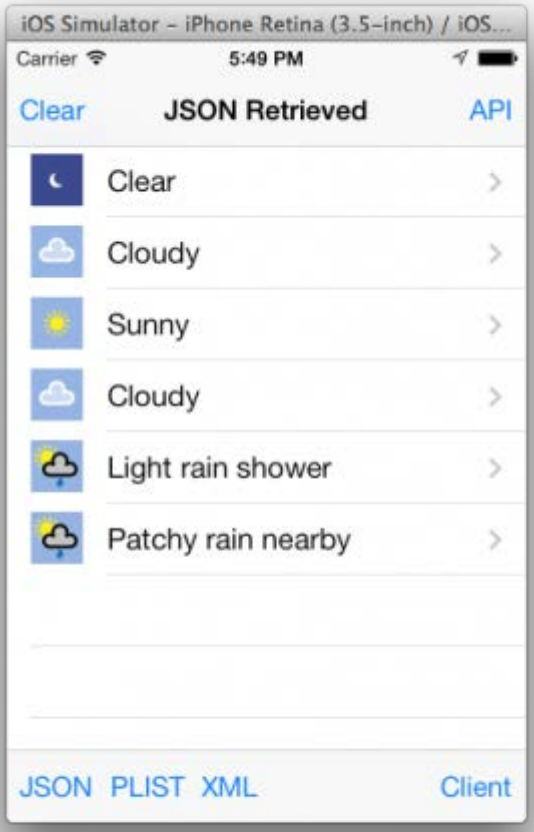
```
    cell.textLabel.text = [daysWeather weatherDescription];

        NSURL *url = [NSURL
URLWithString:daysWeather.weatherIconURL];
        NSURLRequest *request = [NSURLRequest requestWithURL:url];
        UIImage *placeholderImage = [UIImage
imageNamed:@"placeholder"];

        __weak UITableViewCell *weakCell = cell;

        [cell.imageView setImageWithURLRequest:request
                              placeholderImage:placeholderImage
                                       success:^(NSURLRequest
*request, NSHTTPURLResponse *response, UIImage *image) {

                                           weakCell.imageView.image
= image;
                                           [weakCell
setNeedsLayout];

                                       } failure:nil];
```

UIImageView+AFNetworking makes setImageWithURLRequest: and several other related methods available to you.

Both the success and failure blocks are optional, but if you do provide a success block, you must explicitly set the image property on the image view (or else it won't be set). If you don't provide a success block, the image will automatically be set for you.

When the cell is first created, its image view will display the placeholder image until the real image has finished downloading.

Now build and run your project. Tap on any of the operations you've added so far, and you should see this:



Nice! Asynchronously loading images has never been easier.

A RESTful Class

So far you've been creating one-off networking operations using AFHTTPRequestOperation.

Alternatively, AFHTTPRequestOperationManager and AFHTTPSessionManager are designed to help you easily interact with a single, web-service endpoint.

Both of these allow you to set a base URL and then make several requests to the same endpoint. Both can also monitor for changes in connectivity,

encode parameters, handle multipart form requests, enqueue batch operations, and help you perform the full suite of RESTful verbs (GET, POST, PUT, and DELETE).

“Which one should I use?”, you might ask.

If you’re targeting iOS 7 and above, use AFHTTPSessionManager, as internally it creates and uses NSURLSession and related objects.

If you’re targeting iOS 6 and above, use AFHTTPRequestOperationManager, which has similar functionality to AFHTTPSessionManager, yet it uses NSURLConnection internally instead of NSURLSession (which isn’t available in iOS 6). Otherwise, these classes are very similar in functionality.

In your weather app project, you’ll be using AFHTTPSessionManager to perform both a GET and PUT operation.

Note: Unclear on what all this talk is about REST, GET, and POST? Check out this explanation of the subject – What is REST?

Update the class declaration at the top of WTTableViewController.h to the following:

```
@interface WTTableViewController :
UITableViewController<NSXMLParserDelegate,
CLLocationManagerDelegate, UIActionSheetDelegate>
```

In WTTableViewController.m, find the clientTapped: method and replace its implementation with the following:

```
- (IBAction)clientTapped:(id)sender
{
    UIActionSheet *actionSheet = [[UIActionSheet alloc]
initWithTitle:@"AFHTTPSessionManager"
delegate:self
cancelButtonTitle:@"Cancel"
```

```
destructiveButtonTitle:nil
otherButtonTitles:@"HTTP GET", @"HTTP POST", nil];
        [actionSheet showFromBarButtonItem:sender animated:YES];
```

This method creates and displays an action sheet asking the user to choose between a GET and POST request. Add the following method at the end of the class implementation (right before @end) to implement the action sheet delegate method:

```
    - (void)actionSheet:(UIActionSheet *)actionSheet
clickedButtonAtIndex:(NSInteger)buttonIndex
    {
        if (buttonIndex == [actionSheet cancelButtonIndex]) {
            // User pressed cancel -- abort
            return;
        }
        // 1
        NSURL *baseURL = [NSURL URLWithString:BaseURLString];
        NSDictionary *parameters = @{@"format": @"json"};
        // 2
        AFHTTPSessionManager *manager = [[AFHTTPSessionManager alloc]
initWithBaseURL:baseURL];
        manager.responseSerializer = [AFJSONResponseSerializer
serializer];
        // 3
        if (buttonIndex == 0) {
            [manager GET:@"weather.php" parameters:parameters
```

```
success:^(NSURLSessionDataTask *task, id responseObject) {
                self.weather = responseObject;
                self.title = @"HTTP GET";
                [self.tableView reloadData];
            } failure:^(NSURLSessionDataTask *task, NSError *error) {
                UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error Retrieving Weather
message:[error localizedDescription]
delegate:nil
cancelButtonTitle:@"Ok"
otherButtonTitles:nil];
                [alertView show];
            }];
        }
        // 4
        else if (buttonIndex == 1) {
            [manager POST:@"weather.php" parameters:parameters
success:^(NSURLSessionDataTask *task, id responseObject) {
                self.weather = responseObject;
                self.title = @"HTTP POST";
                [self.tableView reloadData];
            } failure:^(NSURLSessionDataTask *task, NSError *error) {
                UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error Retrieving Weather"
message:[error localizedDescription]
delegate:nil
cancelButtonTitle:@"Ok"
otherButtonTitles:nil];
                [alertView show];
```

Here’s what’s happening above:

You first set up the baseURL and the dictionary of parameters.

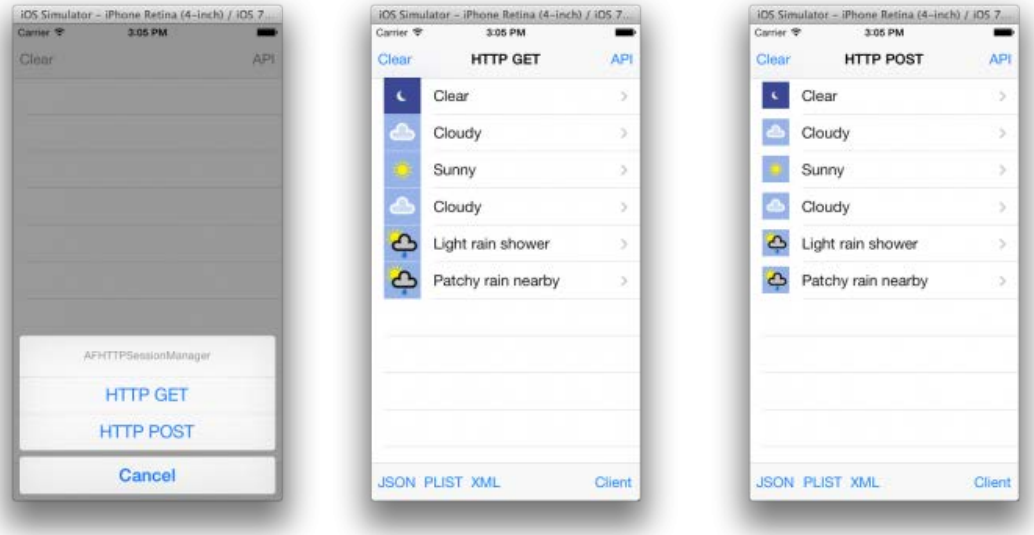
You then create an instance of AFHTTPSessionManager and set its responseSerializer to the default JSON serializer, similar to the previous JSON example.

If the user presses the button index for HTTP GET, you call the GET method on the manager, passing in the parameters and usual pair of success and failure blocks.

You do the same with the POST version.

In this example you’re requesting JSON responses, but you can easily request either of the other two formats as discussed previously.

Build and run your project, tap on the Client button and then tap on either the HTTP GET or HTTP POST button to initiate the associated request. You should see these screens:



At this point, you know the basics of using AFHTTPSessionManager, but there’s an even better way to use it that will result in cleaner code, which you’ll learn about next.

World Weather Online

Before you can use the live service, you'll first need to register for a free account on World Weather Online. Don't worry - it's quick and easy to do!

After you've registered, you should receive a confirmation email at the address you provided, which will have a link to confirm your email address (required). You then need to request a free API key via the My Account page. Go ahead and leave the page open with your API key as you'll need it soon.

Now that you've got your API key, back to AFNetworking…

Hooking into the Live Service

So far you've been creating AFHTTPRequestOperation and AFHTTPSessionManager directly from the table view controller as you needed them. More often than not, your networking requests will be associated with a single web service or API.

AFHTTPSessionManager has everything you need to talk to a web API. It will decouple your networking communications code from the rest of your code, and make your networking communications code reusable throughout your project.

Here are two guidelines on AFHTTPSessionManager best practices:

Create a subclass for each web service. For example, if you're writing a social network aggregator, you might want one subclass for Twitter, one for Facebook, another for Instragram and so on.

In each AFHTTPSessionManager subclass, create a class method that returns a shared singleton instance. This saves resources and eliminates the need to allocate and spin up new objects.

Your project currently doesn't have a subclass of AFHTTPSessionManager; it just creates one directly. Let's fix that.

To begin, create a new file in your project of type iOS\Cocoa Touch\Objective-C Class. Call itWeatherHTTPClient and make it a subclass

of AFHTTPSessionManager.

You want the class to do three things: perform HTTP requests, call back to a delegate when the new weather data is available, and use the user's physical location to get accurate weather.

Replace the contents of WeatherHTTPClient.h with the following:

```
#import "AFHTTPSessionManager.h"

@protocol WeatherHTTPClientDelegate;

@interface WeatherHTTPClient : AFHTTPSessionManager
@property (nonatomic, weak)
id<WeatherHTTPClientDelegate>delegate;

+ (WeatherHTTPClient *)sharedWeatherHTTPClient;
- (instancetype)initWithBaseURL:(NSURL *)url;
- (void)updateWeatherAtLocation:(CLLocation *)location
forNumberOfDays:(NSUInteger)number;

@end

@protocol WeatherHTTPClientDelegate <NSObject>
@optional
-(void)weatherHTTPClient:(WeatherHTTPClient *)client
didUpdateWithWeather:(id)weather;
-(void)weatherHTTPClient:(WeatherHTTPClient *)client
didFailWithError:(NSError *)error;
@end
```

You'll learn more about each of these methods as you implement them. Switch over to WeatherHTTPClient.m and add the following right after the

import statement:

```
    // Set this to your World Weather Online API Key
    static NSString * const WorldWeatherOnlineAPIKey = @"PASTE YOUR
API KEY HERE";

    static NSString * const WorldWeatherOnlineURLString =
@"http://api.worldweatheronline.com/free/v1/";
```

Make sure you replace @" PASTE YOUR KEY HERE" with your actual World Weather Online API Key.

Next paste these methods just after the @implementation line:

```
    + (WeatherHTTPClient *)sharedWeatherHTTPClient
    {
        static WeatherHTTPClient *_sharedWeatherHTTPClient = nil;

        static dispatch_once_t onceToken;
        dispatch_once(&onceToken, ^{
            _sharedWeatherHTTPClient = [[self alloc]
initWithBaseURL:[NSURL URLWithString:WorldWeatherOnlineURLString]];
        });

        return _sharedWeatherHTTPClient;
    }

    - (instancetype)initWithBaseURL:(NSURL *)url
    {
        self = [super initWithBaseURL:url];

        if (self) {
            self.responseSerializer = [AFJSONResponseSerializer
```

```
serializer];
            self.requestSerializer = [AFJSONRequestSerializer
serializer];
        }

        return self;
    }
```

The sharedWeatherHTTPClient method uses Grand Central Dispatch to ensure the shared singleton object is only allocated once. You initialize the object with a base URL and set it up to request and expect JSON responses from the web service.

Paste the following method underneath the previous ones:

```
    - (void)updateWeatherAtLocation:(CLLocation *)location
forNumberOfDays:(NSUInteger)number
    {
        NSMutableDictionary *parameters = [NSMutableDictionary
dictionary];

        parameters[@"num_of_days"] = @(number);
        parameters[@"q"] = [NSString
stringWithFormat:@"%f,%f",location.coordinate.latitude,location.coord
inate.longitude];
        parameters[@"format"] = @"json";
        parameters[@"key"] = WorldWeatherOnlineAPIKey;

        [self GET:@"weather.ashx" parameters:parameters
success:^(NSURLSessionDataTask *task, id responseObject) {
            if ([self.delegate
respondsToSelector:@selector(weatherHTTPClient:didUpdateWithWeather:)
```

```
]) {
                [self.delegate weatherHTTPClient:self
didUpdateWithWeather:responseObject];
            }
        } failure:^(NSURLSessionDataTask *task, NSError *error) {
            if ([self.delegate
respondsToSelector:@selector(weatherHTTPClient:didFailWithError:)]) {
                [self.delegate weatherHTTPClient:self
didFailWithError:error];
            }
        }];
    }
```

This method calls out to World Weather Online to get the weather for a particular location.

Once the object has loaded the weather data, it needs some way to communicate that data back to whoever's interested. Thanks to the WeatherHTTPClientDelegate protocol and its delegate methods, the success and failure blocks in the above code can notify a controller that the weather has been updated for a given location. That way, the controller can update what it is displaying.

Now it's time to put the final pieces together! The WeatherHTTPClient is expecting a location and has a defined delegate protocol, so you need to update the WTTableViewController class to take advantage of this.

Open up WTTableViewController.h to add an import and replace the @interface declaration as follows:

```
#import "WeatherHTTPClient.h"

@interface WTTableViewController : UITableViewController
<NSXMLParserDelegate, CLLocationManagerDelegate,
```

```
UIActionSheetDelegate, WeatherHTTPClientDelegate>
```

Also add a new Core Location manager property:

```
@property (nonatomic, strong) CLLocationManager
*locationManager;
```

In WTTableViewController.m, add the following lines to the bottom of viewDidLoad::

```
self.locationManager = [[CLLocationManager alloc] init];
self.locationManager.delegate = self;
```

These lines initialize the Core Location manager to determine the user's location when the view loads. The Core Location manager then reports that location via a delegate callback. Add the following method to the

```
(void)locationManager:(CLLocationManager *)manager
didUpdateLocations:(NSArray *)locations
{
    // Last object contains the most recent location
    CLLocation *newLocation = [locations lastObject];

    // If the location is more than 5 minutes old, ignore it
    if([newLocation.timestamp timeIntervalSinceNow] > 300)
        return;

    [self.locationManager stopUpdatingLocation];

    WeatherHTTPClient *client = [WeatherHTTPClient
sharedWeatherHTTPClient];
    client.delegate = self;
    [client updateWeatherAtLocation:newLocation
forNumberOfDays:5];
}
```

implementation:

Now when there's an update to the user's whereabouts, you can call the singleton WeatherHTTPClient instance to request the weather for the current location.

Remember, WeatherHTTPClient has two delegate methods itself that you need to implement. Add the following two methods to the implementation:

```
- (void)weatherHTTPClient:(WeatherHTTPClient *)client
didUpdateWithWeather:(id)weather
{
    self.weather = weather;
    self.title = @"API Updated";
    [self.tableView reloadData];
}

- (void)weatherHTTPClient:(WeatherHTTPClient *)client
didFailWithError:(NSError *)error
{
    UIAlertView *alertView = [[UIAlertView alloc]
initWithTitle:@"Error Retrieving Weather"

message:[NSString stringWithFormat:@"%@",error]

delegate:nil

cancelButtonTitle:@"OK" otherButtonTitles:nil];
    [alertView show];
}
```

When the WeatherHTTPClient succeeds, you update the weather data and reload the table view. In case of a network error, you display an error

message.

Find the apiTapped: method and replace it with the following:

```
- (IBAction)apiTapped:(id)sender
{
    [self.locationManager startUpdatingLocation];
}
```

Build and run your project (try your device if you have any troubles with your simulator), tap on the API button to initiate the WeatherHTTPClient request, and you should see something like this:

Here's hoping your upcoming weather is as sunny as mine!

I'm Not Dead Yet!

You might have noticed that this external web service can take some time before it returns with data. It's important to provide your users with feedback when doing network operations so they know the app hasn't stalled or crashed.

Luckily, AFNetworking comes with an easy way to provide this feedback: AFNetworkActivityIndicatorManager.

In WTAppDelegate.m, add this import just below the other:

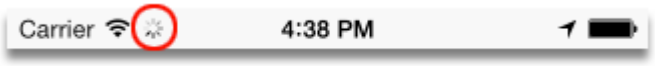
```
#import "AFNetworkActivityIndicatorManager.h"
```

Then find the application:didFinishLaunchingWithOptions: method and replace it with the following:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [AFNetworkActivityIndicatorManager
sharedManager].enabled = YES;
    return YES;
}
```

Enabling the sharedManager automatically displays the network activity indicator whenever a new operation is underway. You won't need to manage it separately for every request you make.

Build and run, and you should see the little networking spinner in the status bar whenever there's a network request:



Now there’s a sign of life for your user even when your app is waiting on a slow web service.

```
- (IBAction)updateBackgroundImage:(id)sender
{
    NSURL *url = [NSURL
URLWithString:@"http://www.raywenderlich.com/wp-content/uploads/2014/
01/sunny-background.png"];
    NSURLRequest *request = [NSURLRequest requestWithURL:url];

    AFHTTPRequestOperation *operation = [[AFHTTPRequestOperation
alloc] initWithRequest:request];
    operation.responseSerializer = [AFImageResponseSerializer
serializer];

    [operation
setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *operation, id
responseObject) {

        self.backgroundImageView.image = responseObject;
        [self saveImage:responseObject
withFilename:@"background.png"];

    } failure:^(AFHTTPRequestOperation *operation, NSError *error)
{

        NSLog(@"Error: %@", error);
```

Downloading Images

If you tap on a table view cell, the app takes you to a detail view of the weather and an animation illustrating the corresponding weather conditions.

That's nice, but at the moment the animation has a very plain background. What better way to update the background than… over the network!

Here's the final AFNetworking trick for this tutorial: AFHTTPRequestOperation can also handle image requests by setting its responseSerializer to an instance of AFImageResponseSerializer.

There are two method stubs in WeatherAnimationViewController.m to implement. Find theupdateBackgroundImage: method and replace it with the following:

This method initiates and handles downloading the new background. On completion, it returns the full image requested.

In WeatherAnimationViewController.m, you will see two helper methods, imageWithFilename: and saveImage:withFilename:, which will let you store and load any image you download. updateBackgroundImage: calls these helper methods to save the downloaded images to disk.

Find the deleteBackgroundImage: method and replace it with the following:

```
- (IBAction)deleteBackgroundImage:(id)sender
{
    NSArray *paths =
NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
    NSString *path = [[paths objectAtIndex:0]
stringByAppendingPathComponent:@"WeatherHTTPClientImages/"];
```

```
        NSError *error = nil;
        [[NSFileManager defaultManager] removeItemAtPath:path
error:&error];

        NSString *desc = [self.weatherDictionary
weatherDescription];
        [self start:desc];
    }
```

This method deletes the downloaded background image so that you can download it again when testing the application.

For the final time: build and run, download the weather data and tap on a cell to get to the detailed view. From here, tap the Update Background button. If you tap on a Sunny cell, you should see this:

Where To Go From Here?

You can download the completed project from here.

Think of all the ways you can now use AFNetworking to communicate with the outside world:

AFHTTPOperation with AFJSONResponseSerializer, AFPropertyListResponseSerializer, or AFXMLParserResponseSerializer response serializers for parsing structured data

UIImageView+AFNetworking for quickly filling in image views

Custom AFHTTPSessionManager subclasses to access live web services

AFNetworkActivityIndicatorManager to keep the user informed

AFHTTPOperation with a AFImageResponseSerializer response serializer for loading images

The power of AFNetworking is yours to deploy!

If you have any questions about anything you've seen here, please pay a visit to the forums to get some assistance. I'd also love to read your comments!

原文链接：http://www.raywenderlich.com/59255/afnetworking-2-0-tutorial

一个用 Arduino 实现的完整项目

介绍

我有一个上小学的女儿. 作为一位父亲，当然该负起责任. 我喜欢自己作为父亲的角色，对此我毫无疑问. 但因此所带给我的困扰则是无止境的数学学习事务. 有时候这真的让我很头疼. 2+2, 5+6, 4+3,一遍又一遍. 那才刚开始呢，现在又多了减法: 5-4, 10-4 .. 而每个人都知道这是没有结束可言的. 我抱怨的够多了，因而决定利用此项技术. 你知道的，技术为人而生，而现在是为我而存在. 到这里来，我亲爱的 Arduino，我需要你.

在最开始的时候，我希望这个项目能很容易实现. 我预计所有需要做的事情就是写一些函数来展示一些数字，并且为了根据趣味性，也许要来点蜂鸣声和一些 LED 灯光. 然后，情况在我开始精心考虑它的时候发生了改变，出现了一些硬件管理问题，然后是内容管理问题. 我们这个小小的 Arduino 应用程序变成了一个认真把握的东西，这导致我写下了这篇文章. 让我们先从需求开始吧.

计划需求

系统可以显示一个菜单，提供一些基本的操作：加、减、乘、除 。

用户（我的女儿）可以用键盘从菜单上选择一种算数运算来学习。

会有一些难度级别：在选择运算后，难度级别会显示出来。

根据选择的难度级别，会随机显示出一些问题，用户可以用键盘回答这些问题。

用户可以在确认前修改自己的答案。

在确认答案后，根据正确与否会显示出一条信息。

如果三次答错，将会显示出正确答案。

用户可以浏览菜单（点开菜单并选择菜单项）。

系统应具有音频和视频警告 API，错误信息可以通过该 API 发送。

每种算术运算有一个限时小测环节。

限时小测随机从简到难给出问题。

测试后会显示出统计数据（回答了多少题目，答对了多少题目）。

在用户接近时系统可以引起她的注意。

可以有一些数学以外的娱乐环节，比如让她唱首歌、亲自己的爸爸等等。如果不这么做，用户将无法继续使用系统。

警告 API 可以用来在娱乐环节做一些有趣的事情。

硬件

我们在这个项目中需要什么：

Arduino Mega

字符液晶面板（Serial LCD）

矩阵键盘

模拟键盘

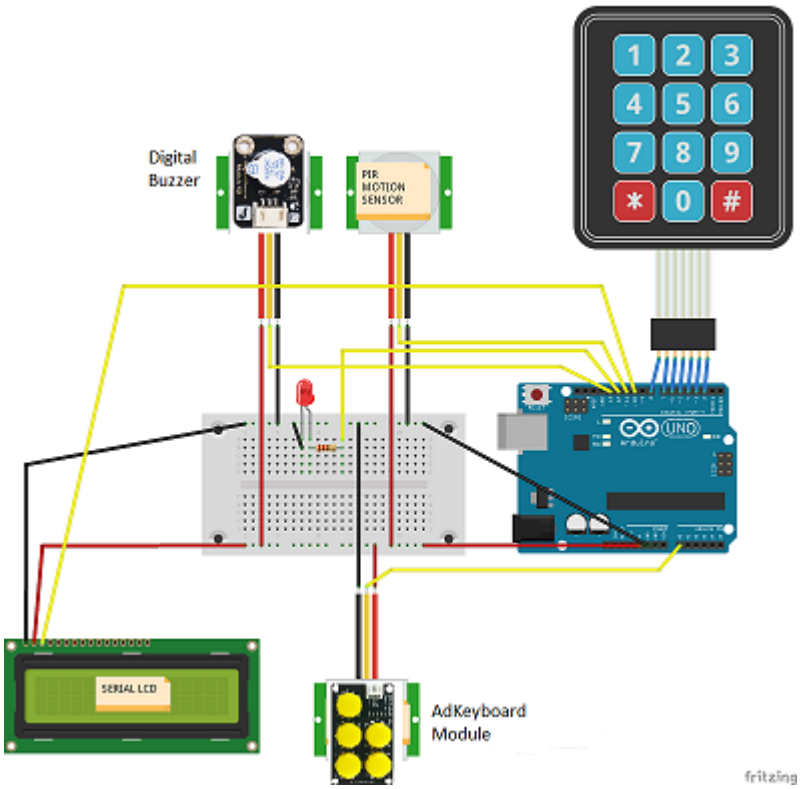
PIR 运动传感器

LED 和 400 欧姆电阻

数字蜂鸣模块

连接线

看看下面的硬件设计：



注意一些和上面不匹配的部分：

LCD 应该是字符 LCD。

Arduino UNO 应该用 Mega 替代。

(我用 Arduino UNO 开启了这个项目，但是因为内存的需要后来改用 Arduino Mega 继续这个项目。开始的时候，Arduino UNO 工作的很好。但是，当代码量增加，我无法将 RAM 使用量控制在 Arduino UNO 的容量之内，然后就想你所想的，我最终启用了 Arduino Mega，它有 8K 的 SRAM。)

软件设计

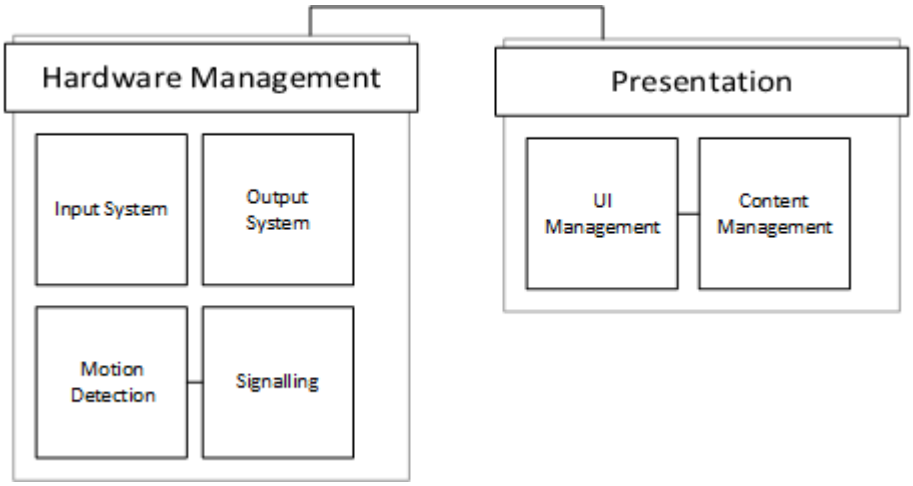


图 1：设计的概览

系统被分为 2 个主要部分。就像你在图 1 中看到的，第一个模块负责硬件的管理。

输入系统：我们有两个不同的键盘，他们被统一在一起，对外提供统一的接口。统一的键盘信息将在（矩阵键盘或模拟键盘上的）任何按键被按下时告知注册的客户端。

输出系统：具有附加功能的字符液晶面板。

发信系统：统一发信子系统。它由一个 LED 和一个数字蜂鸣器组成，它将不同的信号转化为目标客户端的编码。它将不同的信号转换为客户端代码，客户端代码可以根据需求运行各种代码。

运动检测：用 PIR 传感器实现运动检测。当有人被检测到，它触发一个信号来引起注意。

表现层负责与用户的交互。它包含图形界面的处理（菜单和页面）并包含管理子系统。

UI 管理：在这个子系统中，我们定义了图像对象。一个菜单列表被显示出

来供用户选择。一个菜单项可以显示出子菜单或者一个页面。用户可以通过输入显示在其上的索引来选择菜单项。通过按'Escape'键来返回上级菜单。如果一个菜单项是一个页面，选择后将会将这个页面显示出来。页面可以显示出其上面的信息，并等待用户输入来改变它的内容，此时按下'Escape'键就会显示出用户菜单。如果用户输入错误，可以通过按'Backspace'键删除答案。根据答案的正确与否，相应的信号将会被触发。

内容管理：这个子系统提供显示在屏幕上的内容，包括各种算术运算的各种难度级别的生成算法。客户端代码（页面）会向这个子系统请求内容。

简单的类框图如下。这些图像展示了基本的框架，帮助你更容易的理解实际的类实现。

硬件管理

MFK_InputDevice 将 Keypad2 和 AdKeyboard 统一为同一个接口。它处理它们的事件，并向其客户端提供一组新的编码，如下所示。

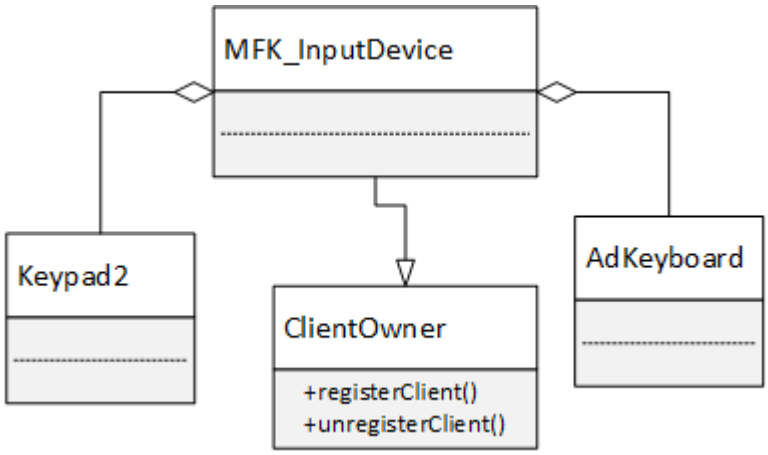


图 2：输入子系统

按键映射：

Keypad	Button	Key	Value (hex)
Matrix	0	'0'	0x30
Matrix	1	'1'	0x31
Matrix	2	'2'	0x32
Matrix	3	'3'	0x33
Matrix	4	'4'	0x34

Matrix	5	'5'	0x35
Matrix	6	'6'	0x36
Matrix	7	'7'	0x37
Matrix	8	'8'	0x38
Matrix	9	'9'	0x39
Matrix	*	Escape	0x1B
Matrix	#	Enter	0x0D
AD	S1	Backspace	0x08
AD	S2	F1	0x80
AD	S3	F2	0x81
AD	S4	F3	0x82
AD	S5	F4	0x83

MFK_OutputDevice 继承自 SerialLCD 类。它结合 SerialLCD 类的功能，并对其进行了增强。

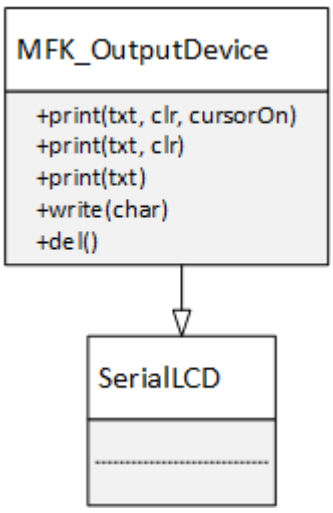


图 3：输出子系统

一个信号模式从信号源产生。一个模式连同它的索引被储存在信号控制器中。想要启动一个信号模式非常简单，只要用它的索引从信号控制器调用它。

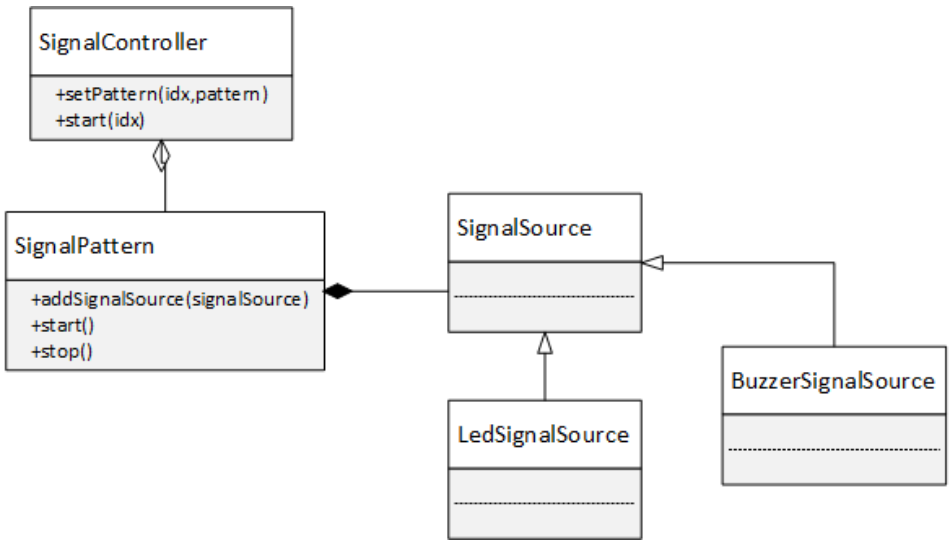


图 4：发信系统

在硬件管理层顶层的是 MFK_Hardware 类。它只会所有其他硬件设备，对客户端隐藏多余的复杂性。举例来说，PIRMotionandSignalController 没有被暴露给客户端。但是输入和输出设备必须要向外界开放，因为 UI 系统需要对这些功能的直接访问。信号模式也是在这个类里构建的，可以通过索引来访问他们。

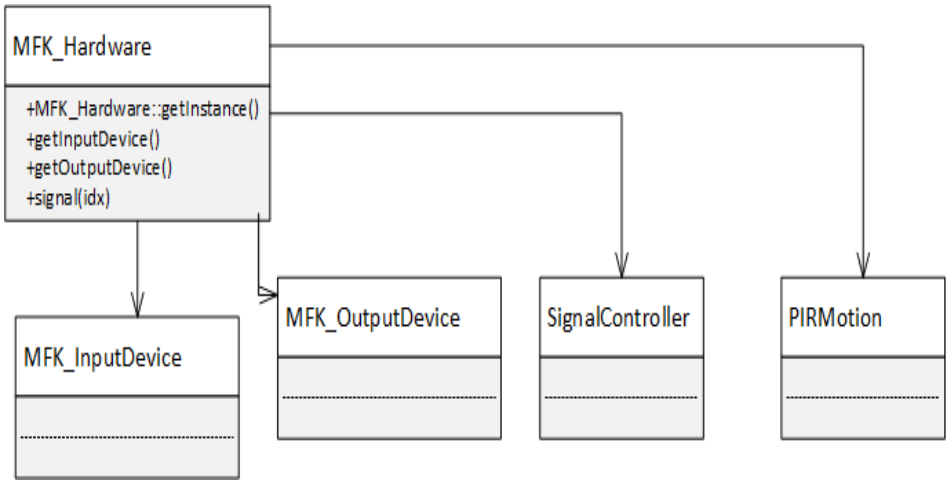


图 5：硬件管理

表现层

这一层负责与用户进行交互，它提供了视觉元素和内容。

ContentFactory 根据 toContentTypeEnum 和 ContentLevelEnum 创建 ContentProviders。客户端得到 ContentFactory 的实例，之后他可以请求一个 content provider。

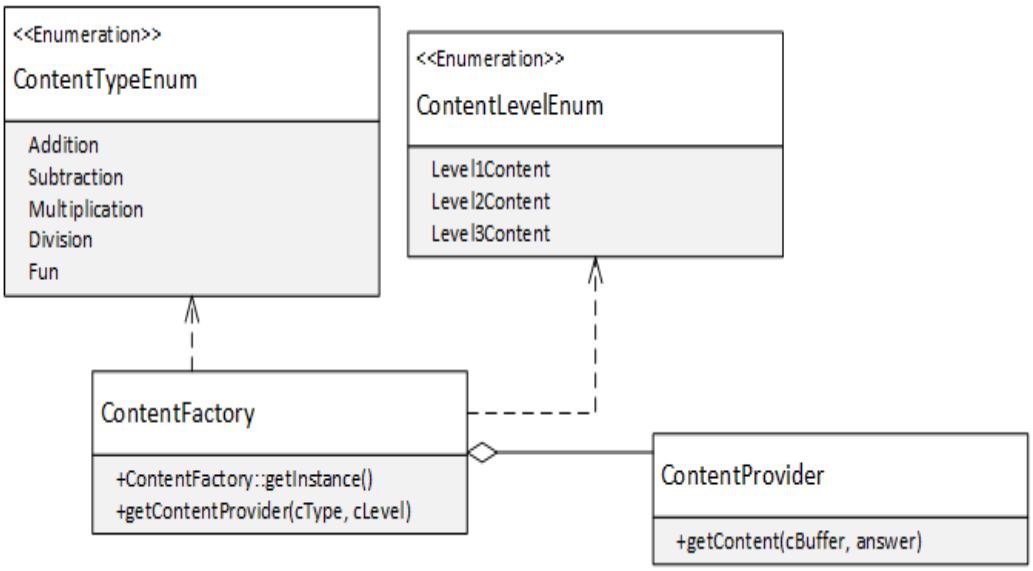


图 6：内容管理

VisualItem 是所有的视觉元素（菜单和页面）的基类。它还将硬件管理和呈现结合起来。'show' 和'msgbox'方法通过调用和回调 VisualItem 提供的方法使用输出设备(MFK_OutputDevice)和输入设备(MFK_InputDevice) 。'msgbox'方法也有能力启动一个信号模式，只要调用硬件(MFK_Hardware)的'signal'方法就可以了。

菜单就像他的名字一样，提供了一系列的菜单项可以选择。用户可以通过一个菜单项前面的索引选择它，然后菜单 'show' VisualItem。

Pageis 是显示内容的视觉工具。除了娱乐内容，它会等待用户的输入。用户'Enter'她的答案，显示信息告知她对错。显示信息之后，就需要从内容管理获取新的内容。

Chapter 是 ContentProvider 和 Page 之间的中间类。当一个页面被第一次显示时，与其相关的 chapter 和 ContentProvider 就会被创建。用户的答案直接由 chapter 处理，并由 chapter 判断其对错。Chapter 也对页面内的学习会话进行统计。FunChapter 是一种不向用户要求答案的 chapter，QuizChapter 是限时的 chapter。在一个 quiz chapter 中，问题只有在时间截止之前才能回答。

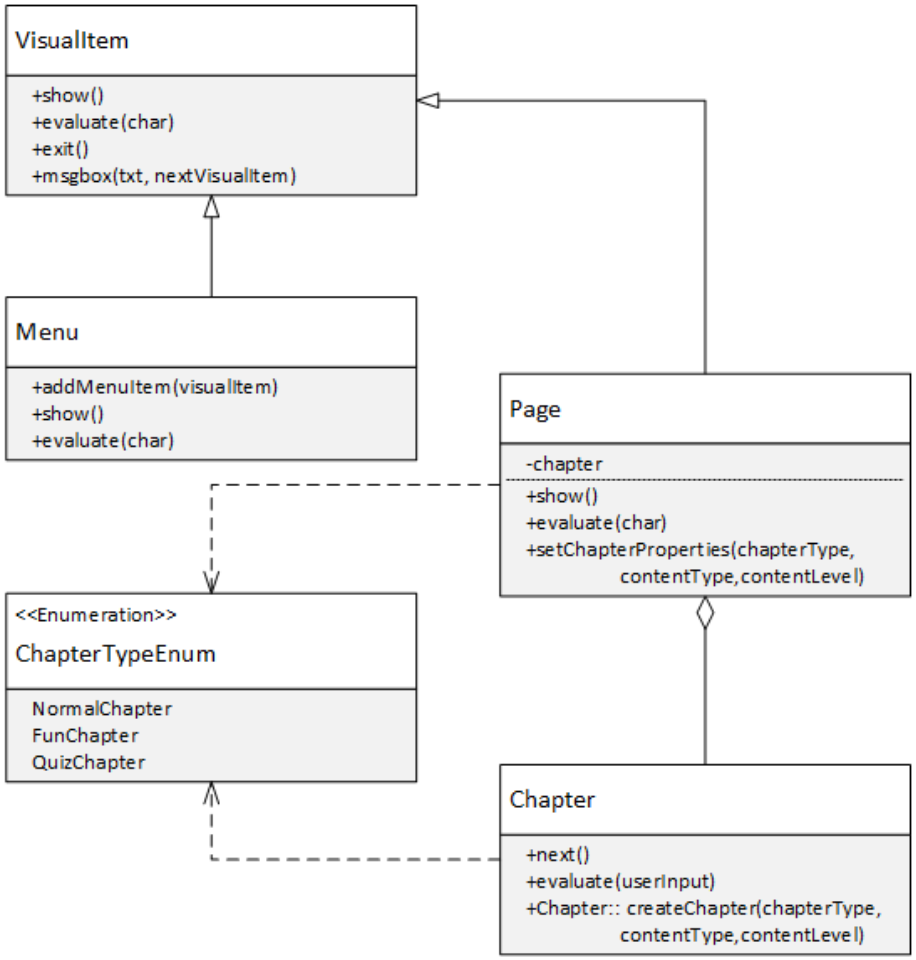


图 7：UI 管理

实现

我希望你已经清楚了系统的通用结构。现在，是时候深入到代码中去，那里是真正的乐趣开始的地方。

我想以 MathForKid.ino 开始。它是上传到 Arduino 主板上的主要代码。

```
01    // File: MathForKid.ino

02    // hardware management

03    MFK_Hardware* hw;

04

05    // presentation

06    Menu* mainMenu;
```

```
07
08      void setup() {
09          // for debugging purposes
10          Serial.begin(9600);
11
12          // get the instance and initialize it
13          hw = MFK_Hardware::getInstance();
14          hw->begin();
15
16          // create user interface
17          CreateUI();
18          // show the main menu
19          mainMenu->show();
20      }
21
22      void loop() {
23          // update hardware
24          hw->update();
25
26          // update active visual item
```

```
27              VisualItem *v = VisualItem::getActiveItem();
28              if(v!=NULL)
29                      v->update();
30      }
```

就这么多。在 Arduino 上运行你的应用吧。好吧，也许解释一下会更好。

正如我在“软件设计”那部分开头所说的，我们有两个部分：一个用来硬件管理，另一个用来展示。它们在代码顶部定义为全局变量，我们在 'setup'函数中将它们初始化。'loop'函数调用代码来更新它们。

事实上，CreateUI 方法也是在这个文件中实现的。当用户开始交互时，它创建用户接口。mainMenu、所有的子菜单和一切页面都是这个方法产生的，chapter 的属性也是其赋予的。

```
01      void CreateUI() {
02              mainMenu = new Menu("main");
03
04              // addition
05              Menu* m = new Menu("+");
06              mainMenu->addMenuItem(m);
07
08              // level-1 page
09              Page* p = new Page("L1");
10              p->setChapterProperties(Chapter::NormalChapter, \
11                              ContentFactory::Addition,
ContentFactory::Level1Content);
```

```
12              m->addMenuItem(p);
13
14              // level-2 page
15              p = new Page("L2");
16              p->setChapterProperties(Chapter::NormalChapter, \
17                                      ContentFactory::Addition,
ContentFactory::Level2Content);
18              m->addMenuItem(p);
19          ...
```

我们接着来看这个应用的设计模式。

就像你所想的，MFK_Hardware 是 Facade 模式的一个例子。它将底层的硬件管理问题隐藏起来，并对客户端提供了干净的接口。它同时也是 Singleton 模式的代表，因为整个系统运行时其只产生一个实例。为了实现这个功能，MFK_Hardware 的构造器、复制和赋值操作都被声明为私有方法。

```
1       // File: MFK_Hardware.h
2       // private constructor to achieve singleton pattern
3               MFK_Hardware();
4               MFK_Hardware(MFK_Hardware const&); // copy disabled
5               void operator=(MFK_Hardware const&); // assigment disabled
```

你只能通过 getInstance 静态方法访问它们，这个方法是公共的：

```
1       // File: MFK_Hardware.h
2       // static method to get the instance
3               static MFK_Hardware* getInstance() {
4                       static MFK_Hardware hw;
```

```
5                   return &hw;

6           };
```

MFK_InputDevice 也是 Facade 模式的一个好例子。它将两个不同的输入设备（矩阵键盘和模拟键盘）映射为同一个设备。不止这样，它还具有 Observer 模式的特征。它是 MFK_InputDeviceClient 类型的 ClientOwner，客户端可以在 MFK_InputDeviceClient 中注册/注销其对 MFK_InputDevice 的监听。这样，当状态变化的时候（在这里就是按键被按下），所有的客户端都会被告知。我在这个项目的很多地方都使用了这个模式。

```
1       // File: MFK_InputDevice.h

2       template<>

3       class MFK_InputDevice<MFK_InputDeviceClient>:

4           public ClientOwner<MFK_InputDeviceClient> {

5       private:

6           ...

7           // informs registered clients

8           void informClients(char);
1         // File: MFK_InputDevice.cpp

2       void MFK_InputDevice<MFK_InputDeviceClient>::informClients(char c) {

3           for(int i=0; i<5; i++) {

4               if(this->client[i]!=NULL)

5                   this->client[i]->invokeMFKInputCallback(c);

6           }

7       }
```

ClientOwner 实现了 register/unregister 功能。一个 MFK_InputDeviceClient 重载 invokeMFKInputCallback 回调方法并将其自己注册到 ClientOwner。之后 MFK_InputDevice 可以通过它提供的回调方法告知它状态的变化。VisualItem 继承了 MFK_InputDeviceClient，因此一个 visual item 可以将自己注册到 MFK_InputDevice 并监听输入。

ContentFactory 类和 ContentProvider 类也有一些有趣的属性。ContentFactory 是一个 Singleton，它也有 Factory 模式的特征。它为客户端产生内容。但 ContentProvider 的输出依赖于客户端的请求。

```
01      // File: ContentFactory.cpp
02      ContentProvider*
ContentFactory::getContentProvider(ContentTypeEnum op,\
03              ContentLevelEnum level) {
04          if(this->contentProvider[op][level] != NULL)
05              return this->contentProvider[op][level];
06
07          ContentProvider *p=NULL;
08
09          switch(op) {
10              case Addition:
11                  switch(level) {
12                      case Level1Content:
13                          p = new ContentProvider(ContentP_0_0);
14                          break;
```

```
15      ...
16              return p;
17      }
```

ContentProvider 类和 Capter 类可以被当做 Strategy 模式的一个例子。一个 content provider 在一个 chapter 环境下运行。ContentProvider 通过为不同实例准备的算法提供信息（在这里就是问题和答案）。

```
1       // File: Chapter.cpp
2       char* Chapter::next() {
3               char *retval=NULL;
4       ...
5               retval = this->contentP[0]->getContent(Chapter::CONTENT, \
6                                       Chapter::ANSWER);
7       ...
8               return retval;
9       }
```

总结

你懂的，开发应用不只是敲代码。好的需求带来好的设计，好的设计带来好的软件。什么是好？这在很多软件工程的书中都有回答（这不是本文的主题）。为了更好地开发软件，我们需要知道设计模式。（顺便问一句，这段中一共出现了多少个好呢？）

设计模式，像它的名字一样，概括了一类问题的解决方案。因此，不要再造轮子了，为了更好的工作，我们需要有关设计模式的基础知识。

原文链接：http://www.oschina.net/translate/a-complete-project-with-arduino

腾讯大规模 Hadoop 集群实践

TDW（Tencent distributed Data Warehouse，腾讯分布式数据仓库）基于开源软件 Hadoop 和 Hive 进行构建，打破了传统数据仓库不能线性扩展、可控性差的局限，并且根据腾讯数据量大、计算复杂等特定情况进行了大量优化和改造。

TDW 服务覆盖了腾讯绝大部分业务产品，单集群规模达到 4400 台，CPU 总核数达到 10 万左右，存储容量达到 100PB；每日作业数 100 多万，每日计算量 4PB，作业并发数 2000 左右；实际存储数据量 80PB，文件数和块数达到 6 亿多；存储利用率 83%左右，CPU 利用率 85%左右。经过四年多的持续投入和建设，TDW 已经成为腾讯最大的离线数据处理平台。

TDW 的功能模块主要包括：Hive、MapReduce、HDFS、TDBank、Lhotse 等，如图 1 所示。TDW Core 主要包括存储引擎 HDFS、计算引擎 MapReduce、查询引擎 Hive，分别提供底层的存储、计算、查询服务，并且根据公司业务产品的应用情况进行了很多深度订制。TDBank 负责数据采集，旨在统一数据接入入口，提供多样的数据接入方式。Lhotse 任务调度系统是整个数据仓库的总管，提供一站式任务调度与管理。

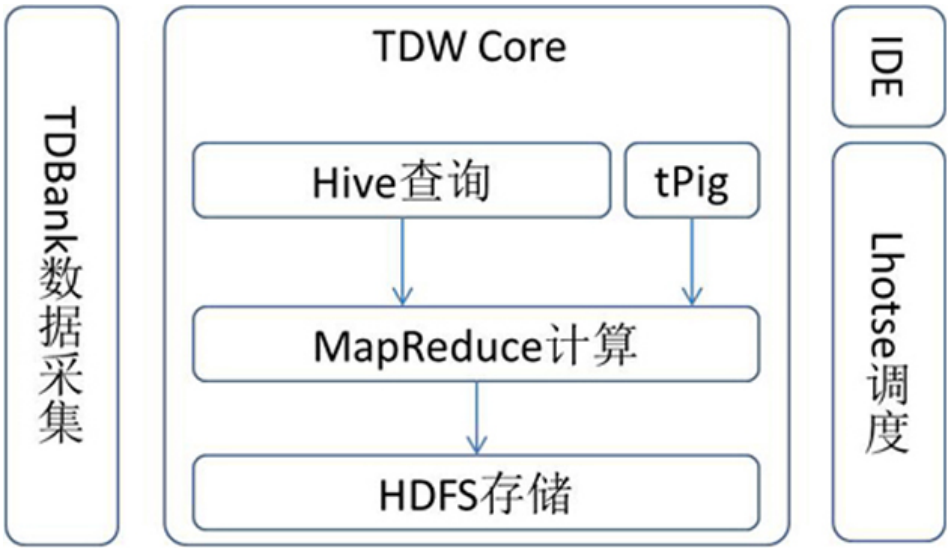


图 1 TDW 的功能模块

建设单个大规模集群的原因

随着业务的快速增长，TDW 的节点数也在增加，对单个大规模 Hadoop 集群的需求也越来越强烈。TDW 需要做单个大规模集群，主要是从数据共享、计算资源共享、减轻运营负担和成本等三个方面考虑。

1. 数据共享。TDW 之前在多个 IDC 部署数十个集群，主要是根据业务分别部署，这样当一个业务需要其他业务的数据，或者需要公共数据时，就需要跨集群或者跨 IDC 访问数据，这样会占用 IDC 之间的网络带宽。为了减少跨 IDC 的数据传输，有时会将公共数据冗余分布到多个 IDC 的集群，这样又会带来存储空间浪费。

2. 计算资源共享。当一个集群的计算资源由于某些原因变得紧张时，例如需要数据补录时，这个集群的计算资源就捉襟见肘，而同时，另一个集群的计算资源可能空闲，但这两者之间没有做到互通有无。

3. 减轻运营负担和成本。十几个集群同时需要稳定运营，而且当一个集群的问题解决时，也需要解决其他集群已经出现的或者潜在的问题。一个 Hadoop 版本要在十几个集群逐一变更，监控系统也要在十几个集群上部署。这些都给运营带来了很大负担。此外，分散的多个小集群，资源利用率不高，机器成本较大。

建设单个大规模集群的方案及优化

面临的挑战

TDW 从单集群 400 台规模建设成单集群 4000 台规模，面临的最大挑战是 Hadoop 架构的单点问题：计算引擎单点 JobTracker 负载重，使得调度效率低、集群扩展性不好；存储引擎单点 NameNode 没有容灾，使得重启耗时长、不支持灰度变更、具有丢失数据的风险。TDW 单点瓶颈导致平台的高可用性、高效性、高扩展性三方面都有所欠缺，将无法支撑 4000 台规模。为了解决单点瓶颈，TDW 主要进行了 JobTracker 分散化和 NameNode 高可用两方面的实施。

JobTracker 分散化

1.单点 JobTracker 的瓶颈

TDW 以前的计算引擎是传统的两层架构，单点 JobTracker 负责整个集群的资源管理、任务调度和任务管理，TaskTracker 负责任务执行。JobTracker 的三个功能模块耦合在一起，而且全部由一个 Master 节点负责执行，当集群并发任务数较少时，这种架构可以正常运行，但当集群并发任务数达到 2000、节点数达到

4000 时，任务调度就会出现瓶颈，节点心跳处理迟缓，集群扩展也会遇到瓶颈。

2.JobTracker 分散化方案

TDW 借鉴 YARN 和 Facebook 版 corona 设计方案，进行了计算引擎的三层架构优化（如图 2 所示）：将资源管理、任务调度和任务管理三个功能模块解耦；JobTracker 只负责任务管理功能，而且一个 JobTracker 只管理一个 Job；将比较轻量的资源管理功能模块剥离出来交给新的称为 ClusterManager 的 Master 负责执行；任务调度也剥离出来，交给具有资源信息的 ClusterManager 负责执行；对性能要求较高的任务调度模块采用更加精细的调度方式。

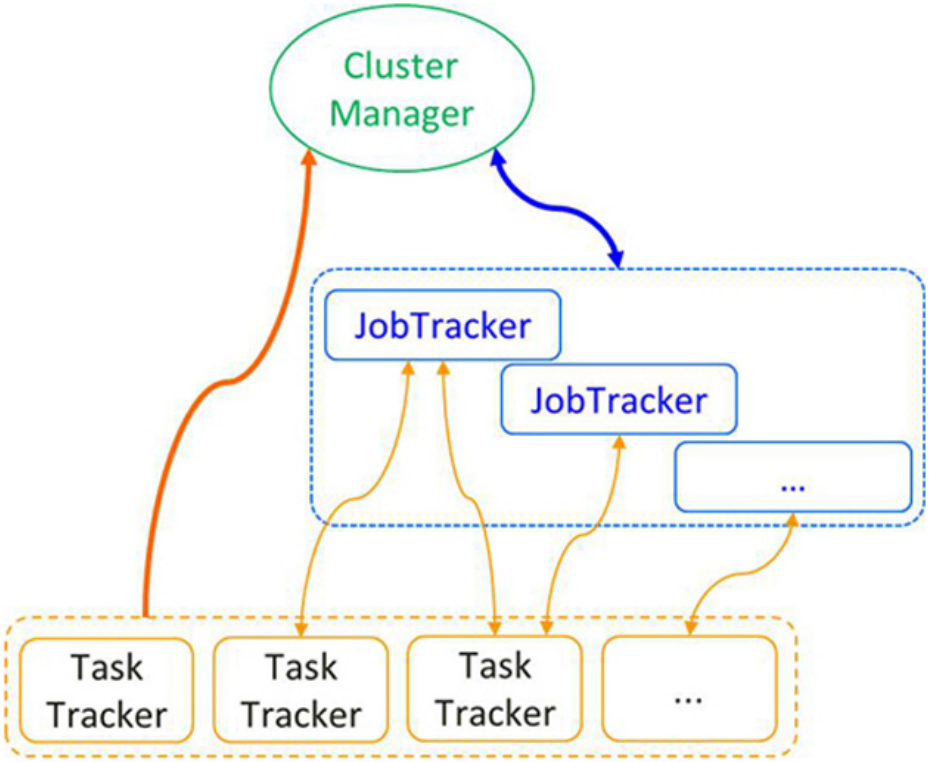


图 2 JobTracker 分散化架构

新架构下三个角色分别是：ClusterManager 负责整个集群的资源管理和任务调度，JobTracker 负责单个 Job 的管理，TaskTracker 负责任务的执行。

（1）两路心跳。之前的架构下，TaskTracker 向 JobTracker 上报心跳，JobTracker 串行地处理这些心跳，心跳处理中进行节点管理、任务管理、任务调度等，心跳繁重，影响任务调度和集群扩展性。新架构下，心跳被拆分成两路心跳，分别上报任务和资源信息。

JobTracker 获知任务信息通过任务上报心跳的方式。任务上报心跳是通过任务所在的 TaskTracker 启动一个新的独立线程向对应的 JobTracker 上报心跳这条途

径，在同一个 TaskTracker 上，不同 Job 的任务使用不同的线程向不同的 JobTracker 上报心跳，途径分散，提升了心跳上报效率。

TaskTracker 通过上报心跳的方式将资源信息汇报给 ClusterManager。ClusterManager 从 TaskTracker 的心跳中获取节点的资源信息：CPU 数量、内存空间大小、磁盘空间大小等的总值和剩余值，根据这些信息判断节点是否还能执行更多的任务。同时，ClusterManager 通过 TaskTracker 与其之间维系的心跳来管理节点的生死存亡。

以前繁重的一路心跳被拆分成了两路轻量的心跳，心跳间隔由 40s 优化成 1s，集群的可扩展性得到了提升。

（2）资源概念。之前架构只有 slot 概念，一般根据核数来设置 slot 数量，对内存、磁盘空间等没有控制。新架构弱化了 slot 概念，加强了资源的概念。

每个资源请求包括具体的物理资源需求描述，包括内存、磁盘和 CPU 等。向 ClusterManager 进行资源申请的有三种来源类型：Map、Reduce、JobTracker，每种来源需要的具体资源量不同。在 CPU 资源上，调度器仍然保留 slot 概念，并且针对三种来源保证各自固定的资源帽。

例如，对于 24 核的节点，配置 13 个核给 Map 用、6 个核给 Reduce 用、1 个核给 JobTracker 用，则认为该节点上有 1 个 JobTracker slot、13 个 Map slot、6 个 Reduce slot。某个 Map 请求的资源需要 2 个核，则认为需要两个 Map slot，当一个节点的 Map slot 用完之后，即使有剩余的 CPU，也不会继续分配 Map 予其执行了。内存空间、磁盘空间等资源没有 slot 概念，剩余空间大小满足需求即认为可以分配。在查找满足资源请求的节点时，会比较节点的这些剩余资源是否满足请求，而且还会优先选择负载低于集群平均值的节点。

（3）独立并发式的下推调度。之前架构下，调度器采用的是基于心跳模型的拉取调度：任务调度依赖于心跳，Map、Reduce 的调度耦合在一起，而且对请求优先级采取全排序方式，时间复杂度为 nlog(n)，任务调度效率低下。

新架构采用独立并发式的下推调度。Map、Reduce、JobTracker 三种资源请求使用三个线程进行独立调度，对请求优先级采取堆排序的方式，时间复杂度为 log(n)。当有资源满足请求时，ClusterManager 直接将资源下推到请求者，而不再被动地等待 TaskTracker 通过心跳的方式获取分配的资源。

例如，一个 Job 有 10 个 Map，每个 Map 需要 1 个核、2GB 内存空间、10GB 磁盘空间，如果有足够的资源，Map 调度线程查找到了满足这 10 个 Map 的节点列表，ClusterManager 会把节点列表下推到 JobTracker；如果 Map 调度线程第一次只查找到了满足 5 个 Map 的节点列表，ClusterManager 会把这个列表下推到 JobTracker，随后 Map 调度线程查找到了剩下 5 个 Map 的节点列表，ClusterManager 再把这个列表下推到 JobTracker。

以前基于心跳模型的拉取调度被优化成独立并发式的下推调度之后，平均调度处理时间由 80ms 优化至 1ms，集群的调度效率得到了提升。

3. Job 提交过程

新架构下，一次 Job 提交过程，需要 Client 和 ClusterManager、TaskTracker 均进行交互（如图 3 所示）：JobClient 先向 ClusterManager 申请启动 JobTracker 所需要的资源；申请到之后，JobClient 在指定的 TaskTracker 上启动 JobTracker 进程，将 Job 提交给 JobTracker；JobTracker 再向 ClusterManager 申请 Map 和 Reduce 资源；申请到之后，JobTracker 将任务启动命令提交给指定的 TaskTracker。

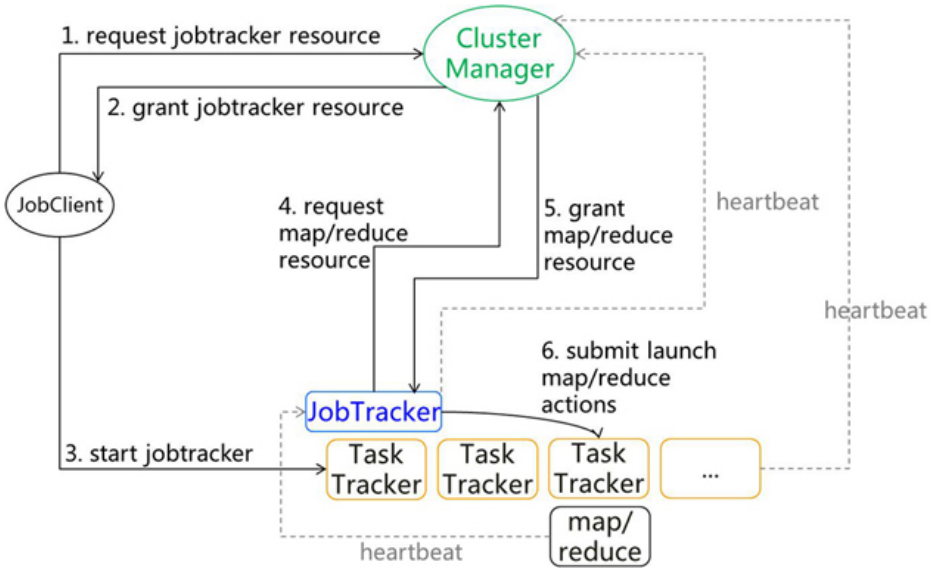


图 3 Job 提交过程

4. 存在的问题及应对措施

JobTracker 分散化方案给计算引擎带来高效性和高扩展性，但没有带来高可用性，单一故障点的问题在此方案中仍然存在，此时的单一故障点问题有别于以前，如下所述。

（1）ClusterManager 如果发生故障，不会造成 Job 状态丢失而且在短时间

内即可恢复。它只存储资源情况，不存储状态，ClusterManager 在很短的时间内可以重启完成。重启之后，TaskTracker 重新向 ClusterManager 汇报资源，ClusterManager 从重启至完全获得集群的资源情况整个阶段可以在 10 秒内完成。

（2）JobTracker 如果发生故障，只会影响单个 Job，对其他 Job 不会造成影响。

基于以上两点，认为新方案的单一故障点问题影响不大，而且考虑方案实施的复杂度和时效性，TDW 在 JobTracker 分散化方案中没有设计高可用方案，而是通过外围系统来降低影响：监控系统保证 ClusterManager 故障及时发现和恢复；Lhotse 调度系统从用户任务级别保证 Job 重试。

NameNode 高可用

1. 单点 NameNode 的问题

TDW 以前的存储引擎是单点 NameNode，在一个业务对应一个集群的情况下，NameNode 压力较小，出故障的几率也较小，而且 NameNode 单点故障带来的影响不会波及全部业务。但当把各个小集群统一到大集群，各个业务都存储之上时，NameNode 压力变大，出故障的几率也变大，NameNode 单点故障造成的影响将会非常严重。即使是计划内变更，停止 NameNode 服务耗时将近 2 个小时，计划内的停止服务变更也给用户带来了较大的影响。

2. NameNode 高可用方案

TDW 设计了一种一主两热备的 NameNode 高可用方案。新架构下 NameNode 角色有三个：一主（ActiveNameNode）两热备（BackupNameNode）。ActiveNameNode 保存 namespace 和 block 信息，对 DataNode 下发命令，并且对客户端提供服务。BackupNameNode 包括 standby 和 newbie 两种状态：standby 提供对 ActiveNameNode 元数据的热备，在 ActiveNameNode 失效后接替其对外提供服务，newbie 状态是正处于学习阶段，学习完毕之后成为 standby。

（1）Replicaton 协议。为了实现 BackupNameNode 对 ActiveNameNode 的元数据一致，随时准备接管 ActiveNameNode 角色，元数据操作日志需要在主备间同步。客户端对元数据的修改不止在 ActiveNameNode 记录事务日志，事务日志还需要从 ActiveNameNode 同步到 BackupNameNode，客户端的每一次写操作，只有成功写入 ActiveNameNode 以及至少一个 BackupNameNode（或者 ZooKeeper）

时，才返回客户端操作成功。当没有 BackupNameNode 可写入时，把事务日志同步到 ZooKeeper 来保证至少有一份事务日志备份。

客户端写操作记录事务日志遵循以下几个原则：

①写入 ActiveNameNode，如果写入失败，返回操作失败，ActiveNameNode 自动退出；

②当写入至少两个节点（Active-NameNode 和 Standby/ZooKeeper/LOG_SYNC newbie）时返回操作成功，其他返回失败；LOG_SYNC newbie 表示 newbie 已经从 ActiveNameNode 获取到全量日志后的状态；

③当只成功写入 ActiveNameNode，此后的 Standby 和 ZooKeeper 均写入失败时，返回失败；

④当只存在 ActiveNameNode 时，进入只读状态。

（2）Learning 协议。newbie 学习机制确保 newbie 启动后通过向 ActiveNameNode 学习获取最新的元数据信息，学习到与 ActiveNameNode 同步时变成 standby 状态。newbie 从 ActiveNameNode 获取最新的 fsimage 和 edits 文件列表，ActiveNameNode 还会和 newbie 之间建立事务日志传输通道，将后续操作日志同步给 newbie，newbie 将这些信息载入内存，构建最新的元数据状态。

（3）事务日志序号。为了验证事务日志是否丢失或者重复，为事务日志指定递增连续的记录号 txid。在事务日志文件 edits 中加入 txid，保证 txid 的连续性，日志传输和加载时保证 txid 连续递增，保存内存中的元数据信息到 fsimage 文件时，将当前 txid 写入 fsimage 头部，载入 fsimage 文件到内存中时，设置元数据当前 txid 为 fsimage 头部的 txid。安全日志序号（safe txid）保存在 ZooKeeper 上，ActiveNameNode 周期性地将 txid 写入 ZooKeeper 作为 safe txid，在 BackupNameNode 转换为 ActiveNameNode 时，需要检查 BackupNameNode 当前的 txid 是否小于 safe txid，若小于则禁止这次角色转换。

（4）checkpoint 协议。新架构仍然具有 checkpoint 功能，以减少日志的大小，缩短重启时构建元数据状态的耗时。由 ActiveNameNode 维护一个 checkpoint 线程，周期性地通知所有 standby 做 checkpoint，指定其中的一个将产生的 fsimage 文件上传给 ActiveNameNode。

（5）DataNode 双报。Block 副本所在的节点列表是 NameNode 元数据信息

的一部分，为了保证这部分信息在主备间一致性，DataNode 采用双报机制。DataNode 对块的改动会同时广播到主备，对主备下发的命令，DataNode 区别对待，只执行主机下发的命令而忽略掉备机下发的命令。

（6）引入 ZooKeeper。主要用来做主节点选举和记录相关日志：NameNode 节点状态、安全日志序号、必要时记录 edit log。

3. 主备切换过程

当主退出时主备状态切换的过程（如图 4 所示）：当 ActiveNameNode 节点 IP1 由于某些原因退出时，两个备节点 IP2 和 IP3 通过向 ZooKeeper 抢锁竞争主节点角色；IP2 抢到锁成为 ActiveNameNode，客户端从 ZooKeeper 上重新获取主节点信息，和 IP2 进行交互，这时即使 IP1 服务恢复，也是 newbie 状态；事务日志在主备间同步，newbie IP1 通过向主节点 IP2 学习成为 standby 状态。

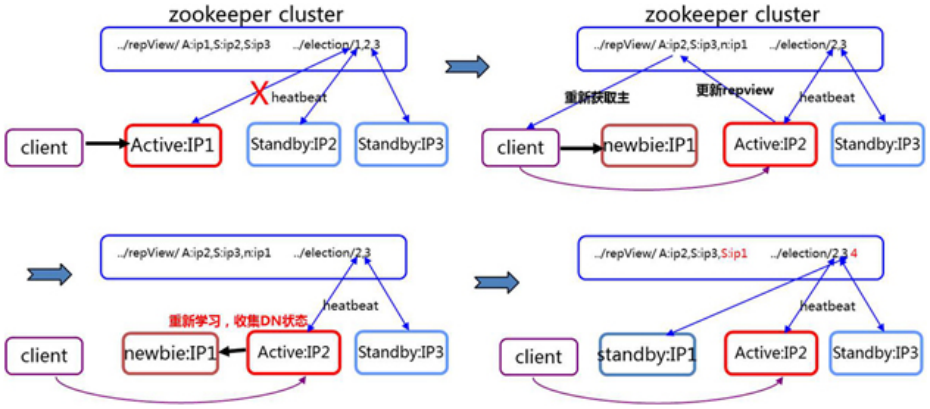


图 4 主退出时主备状态切换

4. 存在的问题

NameNode 高可用方案给存储引擎带来了高可用性，但在高效性方面做出了一些牺牲，由于事务日志需要同步，写性能有 20%左右的下降。

其他优化

TDW 在实施大集群过程中，除了主要实施 JobTracker 分散化和 NameNode 高可用两个方案，还进行了一些其他优化。

1. NameNode 分散化

随着存储量和业务的不断增长，一个 HDFS 元数据空间的访问压力与日俱增。通过 NameNode 分散化来减少一个元数据空间的访问压力。NameNode 分散化主要对元数据信息进行分拆，对用户透明，用户访问认为处于同一个存储引擎，底层可以拆分成多个集群。TDW 在 Hive 层增加用户到 HDFS 集群的路由表，用户

表的数据将写入对应的 HDFS 集群，对外透明，用户只需使用标准的建表语句即可。TDW 根据公司业务的实际应用场景，根据业务线和共享数据等把数据分散到两个 HDFS 集群，有利于数据共享同时也尽量规避集群间的数据拷贝。采用简单、改动最少的方案解决了实际的问题。

2. HDFS 兼容

TDW 内部有三个 HDFS 版本：0.20.1、CDH3u3、2.0，线上主流版本是 CDH3u3，主流 HDFS 版本使用的 RPC 框架尚未优化成 Thrift 或者 Protocol Buffers 等，三个版本互不兼容，增加了互相访问的困难。通过 RPC 层兼容方式实现了 CDH3u3 和 0.20.1 之间的互通，通过完全实现两套接口方式实现了 CDH3u3 和 2.0 之间的互通。

3. 防止数据误删除

重要数据的误删除会给 TDW 带来不可估量的影响，TDW 为了进一步增加数据存储可靠性，不仅开启 NameNode 回收站特性，还增加两个特性： 删除黑白名单，删除接口修改成重命名接口，白名单中的目录可以被删除，白名单中的 IP 可以进行删除操作，其他则不可；DataNode 回收站，块删除操作不会立即进行磁盘文件的删除，而是维护在待删除队列里，过期之后才进行实际的删除操作，这样可以保证在一定时间内如果发现重要的数据被误删除时可以进行数据恢复，还可以防止 NameNode 启动之后元数据意外缺失而造成数据直接被删除的风险。

结语

TDW 从实际情况出发，采取了一系列的优化措施，成功实施了单个大规模集群的建设。为了满足用户日益增长的计算需求，TDW 正在进行更大规模集群的建设，并向实时化、集约化方向发展。TDW 准备引入 YARN 作为统一的资源管理平台，在此基础上构建离线计算模型和 Storm、Spark、Impala 等各种实时计算模型，为用户提供更加丰富的服务。

原文链接：http://www.csdn.net/article/2014-02-19/2818473-Tencent-Hadoop

日 800 万访客、20 万 RPS 网站的 5 个 9 可用性架构

当下，AOL.com 的架构已发展到第五代，也可以说是 20 年内重建了 5 次；现在使用的架构是在 6 年前设计，虽然整体保持不变，但组件的更新和添加却从未停止。在 6 年的持续改进过程中，代码、工具、开发、部署等环节都得到了充分的调优，也让 AOL.com 在当下的数据洪流中屹立不倒。

AOL.com 工程团队一直保持在 25 人左右，包括了开发、测试及运维人员，公司的主要工作地点在 Dulles 和 Virginia，也有一小部分在 Dublin 及 Ireland。应对如此流量并将可用性保持在 5 个 9 从来都不是件容易的事情，在系统架构中我们使用了多种技术：Java、JavaServer Pages、Tomcat、Apache、CentOS、Git、Jenkins、Selenium 和 jQuery 等。

设计原则

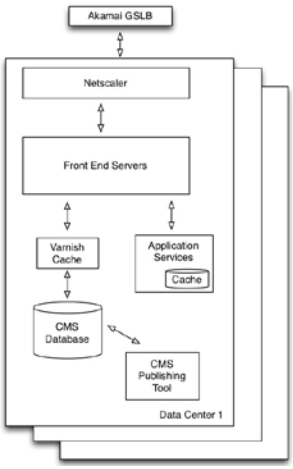
关于架构的设计我们有着明确的思路，其中最重要的就是冗余一切，在系统中某个部分发生故障，或者需要离线维护时，有个备份无疑省时省力，而 5 个 9 的可用率要求每年不超过 5 分钟宕机时间。

第二个原则就是 AOL.com 不能依赖任何共享基础设施去交付页面，即使某个系统或内容发生故障，AOL.com 仍然需要维持着高可用。在这个架构设计后不久，AOL 大部分的网络内容都共享一个被称为 Big Bowl 的基础设施，这样会存在一个非常致命的问题——不同内容之间的相互影响。为了解决这个问题，当下的 AOL 专门设计了不同内容之间的隔离，任何依赖 AOL.com 的内容都会被一个保护服务前移至一组更少的主机上，保护服务负责将调用聚集到下游系统。因此，取代从上万个服务器上接收请求，下游系统可能只会从 20 个不同的服务器上获取请求，响应同样会被缓存来减少负载。同时，运维团队还会对 AOL.com 备份外部数据库。这样一来，在整个系统中只有网络和协议服务被共享。

物理基础设施

AOL.com 部署在 3 个不同的数据中心，两个在 Northern Virginia，一个在 California，这些数据中心都由公司自主运营。虽然每个数据中心的规模都足以支撑整个 AOL.com，但是 AOL.com 仍然坚持同时运行这三个数据中心，多冗余让数据中心的离线维护变得简单。

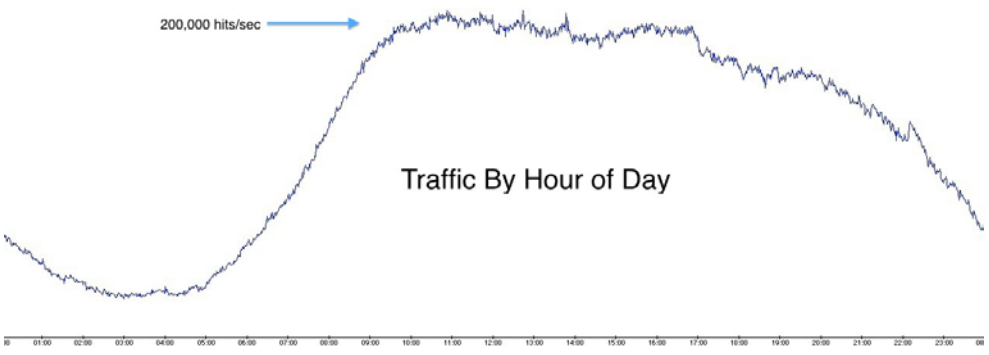
当请求接入时，负责跨数据中心负载均衡的 Akamai GSLB 将为用户指向离他最近的数据中心。为静态内容使用了 Akamai CDN，一旦确定某个数据中心，进一部的请求（数据库或者是服务）都会传入这个数据中心。用户的会话信息会保存在 cookies 里，并通过请求发送；因为不需要保存状态，所以请求可以在任何服务器上被执行。



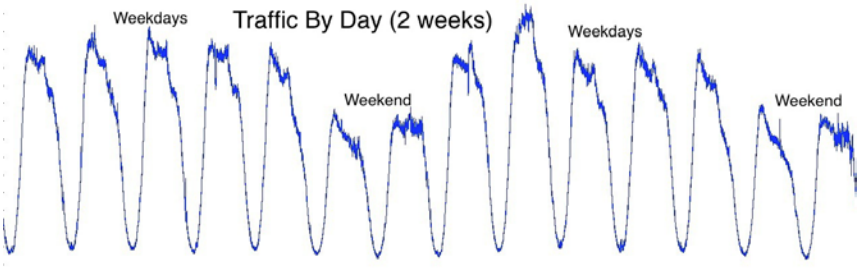
在数据中心上，请求会被发送给 Netscaler 组件，并通过负载均衡的方式将请求传送给前端应用服务器，当下 3 个数据中心前端服务器的数量已接近 700。鉴于所有前端都是虚拟服务器，运维人员可以根据需要快速的添加容量，每个服务器配备了 2 个虚拟 CPU、4GB RAM 和 80GB 的磁盘空间。每个前端服务器都单独运行了 Apache 和 Tomcat，Apache 会被请求发送给本地主机上的 Tomcat，而 Tomcat 则负责大多数请求的处理，调用数据库或者服务以及执行应用程序逻辑。

流量

AOL.com 的流量遵循常见的互联网使用模式——通常情况下流量不会有太大的变化，当然现实世界中发生大事件时除外。



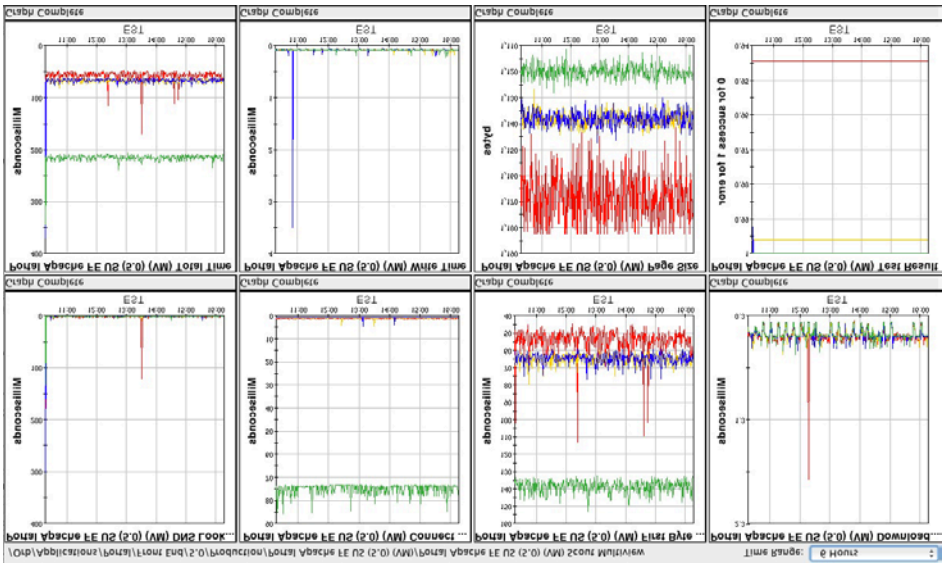
每天流量最低的时候是早上 3-6 点，10 点之前则是一个流量剧增期，会持续 5 个小时保持在 20 万点击每秒，在下午 5 点后降低。在一般情况下，工作日的流量会高于周末。



这些流量由个数据中心共同支撑，两个东海岸的数据中心各负责 40%，西海岸则负责剩余的 20%，流量分布不均匀的原因归结于用户分布的不均匀，同时也因为加拿大、英国、法国等国家的用户也被路由给了东海岸数据中心。

监视

所有的应用程序都运行在 AOL 数据中心，包括 AOL.com，都由一个自主研发的工具监控，类似于 Amazon 的 CloudWatch，已投入使用多年。监视工具会实时的收集软硬件信息，并通过一个客户端应用程序提供报告、图及仪盘表，提供了主机、CPU、接口、网络设备、文件系统、网络流量、响应代码、响应时间、应用程序度量等众多信息。服务器端点更是每分钟都进行检查，并在超过基于可用性及响应时间设置的阈值时进行报警。

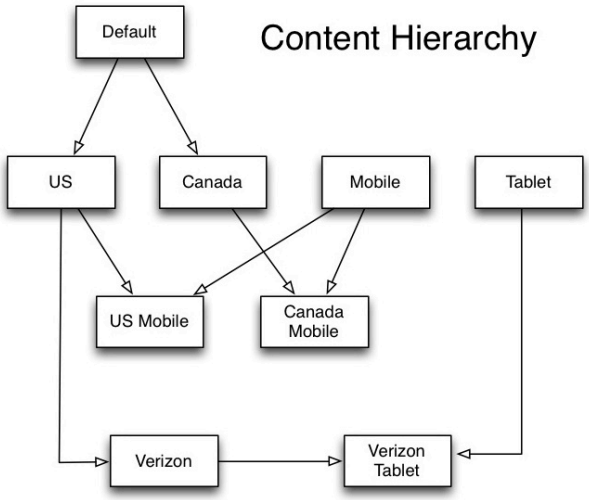


内容管理系统

大部分的 AOL.com 内容及许多的业务逻辑都来自 Content Management

System，CMS 同样是个自主研发系统，建立在相同的 Java/JSP 上，它的功能远超越一般的 CMS。编辑使用它来创建 AOL.com 上的可见内容，开发者使用它来配置应用程序；它同样还是个仪表盘，让编辑可以实时的了解页面上所有部分的执行情况。

同样，AOL 的首页远不止单一域名 www.aol.com 上的一个页面。它其实由不同域名商的数十个版本组成，它们之间可能存在明显或细微的区别。CMS 允许这些不同的版本在同一个地方制作，任何一个版本都可以从相同层上的多个父类继承不同内容，版本间的区别可以包括不同页面上的品牌化 Logo、不同 ID 造成的内容不同等等，比如因为国际化的和访问设备造成的不同。



因此，内容管理系统减少了或者消除了手动的复制粘贴。比如，某个突发的事情只与美国相关，那么编辑只会在美国接口上呈现这个页面，然后这个内容将传播到美国网站的所有份数网站，同样也可以实现为自助的传播到下行网站。

在投放前，我们曾给这个项目进行了一定的尝试。为了更好的测试，我们建立了多变量、多单元并行测试工具。持之以恒的进行测试，并找出优化的途径。首先，我们选择一定比例的测试受众，他们将访问一个不同的版本（可能只是按钮颜色不同，也可能是完全不同的浏览体验），通过浏览器 cookie 进行追踪，通常往往会测试好几周才会确定一个结果。

数据库

AOL.com 上的内容是高度动态的，因此需要为每次页面访问都制定数据库和

应用程序连接规则。除了在页面上提示，CMS 同样包含了许多规则和条件内容，如果你在旧浏览器上，你可能会在顶部看到一个浏览器升级提示。因此，CMS 数据需要足够的快，有能力处理流量的爆发，并且一直可用。我们当下使用的是 MySQL 5，区别于前端的虚拟服务器，数据库服务器拥有更多的容量——16 CPU 的物理服务器。

CMS 数据库使用了 30 个从副本，每个数据中心 10 个；同时，我们还会在一个数据中心建立主节点，同时还建立这个主节点的备份节点。除了主从设计之外，还会在每个数据中心设立 1 个中继器（reapeater）——主节点的另一个备份，负责与整个数据中心的从节点通信，repeater 的作用是减少跨数据中心的数据通信；同时，在这种情况下，如果主节点和其备份节点都宕掉，中继器中的 1 个将会被指派为新的主节点。

应用程序通过 1 个 HTTP 接口访问数据库，AOL 还为 MySQL 开发了 1 个 Apache 模块。每个数据库主机都有一个安装了 Apache 模块的 Web 服务器，它们负责管理数据库的连接池，给 GET 请求做 SQL 查询，结果则以 XML 格式返回。对于 AOL.com 这样的 Java 应用程序，还会有一个 Java 客户端，负责抽象 HTTP 调用并将 XML 解析成对象类型。

之所以使用 HTTP 数据库接口有多个原因：首先，它让客户端可以更轻松的访问数据库，因为任何语言都可以做 HTTP 调用，开发者不必再去考虑 MySQL 客户端驱动及连接池；其次，它有利于应用程序扩展——应用程序通过指定的 URL 访问数据库，URL 对应负载局衡器的 IP 地址，当新的从节点加入时候，客户端应用程序不需要进行重新配置。此外，使用 HTTP 接口还有利于监视。标准的 Web 服务器访问日志和监视工具可以提供数据库事务量、查询次数及错误，然而在查询包含了 URL 作为参数后，日志将变得更加明了；同时，因为 Apache 模块中还加入了管理员接口，因此运维人员可以在任何浏览器上获取数据库状态。

缓存

AOL 的架构中多处使用了缓存，而在 CMS 中就有两个级别的缓存。第一，CMS 中访问数据的 Java 代码使用了内存缓存。鉴于 CMS 中的内容片都被版本化了，只要是新版本的话就不会有什么改动，因此数据可以被缓存来减少数据库 IO。这种缓存是在 Tomcat 实例级别，每 700 实例使用 1 个单独的缓存。

但是为了得知是否有新的版本加入，我们仍然需要时常的查询数据库，这将带来大量的数据库 IO，通常情况下返回的还是相同的结果。鉴于我们的数据库查询都由 HTTP 接口通过 Apache 模块完成，使用 Varnish Cache 来缓存查询将异常简单。数据库查询都是非常简单的 HTTP GET 请求，使用了 URL 做参数的完全 SQL，因此 Varnish 显著的减少了访问数据库服务器的流量。

Akamai CDN 被用于缓存所有的静态内容，除了静态内容之外，AKmai 每隔几秒还会缓存 AOL.com 的静态版本，这个副本被用于极端情况下的灾难恢复——所有数据中心都不可访问时，用户将直接访问这个副本的类容，直到数据中心恢复。

系统中最后一处缓存用于 AOL.com 前端 JSP 代码，前端代码的作用是从 CMS 收集众多的页面并将它们聚合到 HTML。我们开发 JSP 标签库让开发者可以缓存聚合 HTML 所需的任何部分，比如指定页面的那个部分需要缓存只需要用 <cache:cache> </cache:cache>这对标签包含它们。

```
<%@ taglib uri="/WEB-INF/genericcache.tld" prefix="cache"%>

<!-- This part is retrieved from the CMS every time -->
<h1>${title}</h1>

<!-- This part is cached for 10,000 ms -->
<cache:cache template="module_name" objectId="${some_unique_id}" duration="10000">
    <c:forEach var="i" begin="1" end="5">
       Cached Item  <c:out value="${i}"/><p>
    </c:forEach>
</cache:cache>
```

开发过程

大部分的时间，AOL 团队都遵循 Scrum 开发过程，但是也不乏因为业务需求需要加班加点的时候。大部分的网站修改都可以通过 CMS 完成，避开了建立或代码开发过程。这种类型的更新每天可能发生数起至数十起，耗费的时间也是几分钟到几天不等。

开发团队会订阅一份 iPhone 周刊来监视应用程序状态，一旦发现异常数据，比如下游系统丢失，因为冗余策略虽然不会立刻影响到终端用户，但是却是在事情进一步恶化之前的提醒。除了应用程序方面，运维团队在网络、主机、数据库问题上都建立了类似的机制，让团队可以应对任何状况。

开发者在局部环境工作，大部分都使用装有 Netbeans 或 Eclipse 的 Macbook Pro 笔记本。在开发过程中存在 6 个不同的生产环境——与生产环境配置相同，但是规模较小。

代码在发布前需要经过严格的 QA 过程，同时鉴于 AOL.com 的多版本，测试环节所需要做的事情更加复杂（详情见原文）。

回顾和展望

当下 AOL.com 的技术堆栈已非常成熟，鉴于系统的架构设计于 6 年前，部分设计在今天可能已经不是第一选择，比如 Java/Tomcat/JSP 已经给 Python 和 PHP 应用让路，Apache 可能也会略逊于 Nginx，同时 NoSQL 数据库也带来了更多的选择，许多选择已运用在 AOL 的其它系统中。

不能免俗的是，架构中带来灵活和稳定的部分同样阻碍了新技术的采用，然而这并不能阻碍我们给系统做出有益的改变，比如：从实体到虚拟机、添加 Varnish Cache 以及引入 Jenkins。同时，我们现在还在重写前端的 HTML 和 CSS 以清理历史遗留代码，同时也有助于多年前不可能关注的响应速度。总而言之，根据需求不停的演变架构以迎合产业的需求才是关键所在。

原文链接：

http://www.csdn.net/article/2014-02-21/2818488-how-the-aolcom-architecture-evolved-to-99999-availability

Redis 到底有多快[译文]

Redis 自带了一个叫 redis-benchmark 的工具来模拟 N 个客户端同时发出 M 个请求。 （类似于 Apacheab 程序）。你可以使用 redis-benchmark -h 来查看基准参数。

以下参数被支持：

```
Usage: redis-benchmark [-h <host>] [-p <port>] [-c <clients>] [-n <requests]> [-k <boolean>]

 -h <hostname>      Server hostname (default 127.0.0.1)
 -p <port>          Server port (default 6379)
 -s <socket>        Server socket (overrides host and port)
 -c <clients>       Number of parallel connections (default 50)
 -n <requests>      Total number of requests (default 10000)
 -d <size>          Data size of SET/GET value in bytes (default 2)
 -k <boolean>       1=keep alive 0=reconnect (default 1)
 -r <keyspacelen>   Use random keys for SET/GET/INCR, random values for SADD
  Using this option the benchmark will get/set keys
  in the form mykey_rand:000000012456 instead of constant
  keys, the <keyspacelen> argument determines the max
  number of values for the random number. For instance
  if set to 10 only rand:000000000000 - rand:000000000009
  range will be allowed.
 -P <numreq>        Pipeline <numreq> requests. Default 1 (no pipeline).
 -q                 Quiet. Just show query/sec values
 --csv              Output in CSV format
 -l                 Loop. Run the tests forever
```

-t <tests> Only run the comma separated list of tests. The test

names are the same as the ones produced as output.

-I Idle mode. Just open N idle connections and wait.

你需要在基准测试之前启动一个 Redis 实例。一般这样启动测试：

redis-benchmark -q -n 100000

这个工具使用起来非常方便，同时你可以使用自己的基准测试工具， 不过开始基准测试时候，我们需要注意一些细节。

只运行一些测试用例的子集

你不必每次都运行 redis-benchmark 默认的所有测试。 使用-t 参数可以选择你需要运行的测试用例，比如下面的范例：

$ redis-benchmark -t set,lpush -n 100000 -q

SET: 74239.05 requests per second

LPUSH: 79239.30 requests per second

在上面的测试中，我们只运行了 SET 和 LPUSH 命令， 并且运行在安静模式中（使用-q 参数）。

也可以直接指定命令来直接运行，比如下面的范例：

$ redis-benchmark -n 100000 -q script load "redis.call('set','foo','bar')"

script load redis.call('set','foo','bar'): 69881.20 requests per second

选择测试键的范围大小

默认情况下面，基准测试使用单一的 key。在一个基于内存的数据库里， 单一 key 测试和真实情况下面不会有巨大变化。当然，使用一个大的 key 范围空间， 可以模拟现实情况下面的缓存不命中情况。

这时候我们可以使用-r 命令。比如，假设我们想设置 10 万随机 key 连续 SET 100 万次，我们可以使用下列的命令：

```
$ redis-cli flushall
OK
$ redis-benchmark -t set -r 100000 -n 1000000
====== SET ======
  1000000 requests completed in 13.86 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1
99.76% `<=` 1 milliseconds
99.98% `<=` 2 milliseconds
100.00% `<=` 3 milliseconds
100.00% `<=` 3 milliseconds
72144.87 requests per second
$ redis-cli dbsize
(integer) 99993
```

使用 pipelining

默认情况下，每个客户端都是在一个请求完成之后才发送下一个请求（benchmark 会模拟 50 个客户端除非使用-c 指定特别的数量）， 这意味着服务器几乎是按顺序读取每个客户端的命令。Also RTT is payed as well.

真实世界会更复杂，Redis 支持/topics/pipelining，使得可以一次性执行多条命令成为可能。 Redis pipelining 可以提高服务器的 TPS。

下面这个案例是在 Macbook air 11" 上使用 pipelining 组织 16 条命令的测试范例：

```
$ redis-benchmark -n 1000000 -t set,get -P 16 -q
SET: 403063.28 requests per second
GET: 508388.41 requests per second
```

记得在多条命令需要处理时候使用 pipelining。

陷阱和错误的认识

第一点是显而易见的：基准测试的黄金准则是使用相同的标准。 用相同的

任务量测试不同版本的 Redis,或者用相同的参数测试测试不同版本 Redis。如果把 Redis 和其他工具测试，那就需要小心功能细节差异。

Redis 是一个服务器：所有的命令都包含网络或 IPC 消耗。这意味着和它和 SQLite， Berkeley DB， Tokyo/Kyoto Cabinet 等比较起来无意义， 因为大部分的消耗都在网络协议上面。

Redis 的大部分常用命令都有确认返回。有些数据存储系统则没有（比如 MongoDB 的写操作没有返回确认）。把 Redis 和其他单向调用命令存储系统比较意义不大。

简单的循环操作 Redis 其实不是对 Redis 进行基准测试,而是测试你的网络（或者 IPC）延迟。想要真正测试 Redis，需要使用多个连接（比如 redis-benchmark)， 或者使用 pipelining 来聚合多个命令，另外还可以采用多线程或多进程。

Redis 是一个内存数据库，同时提供一些可选的持久化功能。 如果你想和一个持久化服务器（MySQL, PostgreSQL 等等） 对比的话， 那你需要考虑启用 AOF 和适当的 fsync 策略。

Redis 是单线程服务。它并没有设计为多 CPU 进行优化。如果想要从多核获取好处， 那就考虑启用多个实例吧。将单实例 Redis 和多线程数据库对比是不公平的。

一个普遍的误解是 redis-benchmark 特意让基准测试看起来更好，所表现出来的数据像是人造的，而不是真实产品下面的。

Redis-benchmark 程序可以简单快捷的对给定硬件条件下面的机器计算出性能参数。但是,通常情况下面这并不是 Redis 服务器可以达到的最大吞吐量。事实上,使用 pipelining 和更快的客户端(hiredis)可以达到更大的吞吐量。redis-benchmark 默认情况下面仅仅使用并发来提高吞吐量（创建多条连接)。它并没有使用 pipelining 或者其他并行技术(仅仅多条连接,而不是多线程)。

如果想使用 pipelining 模式来进行基准测试（了达到更高吞吐量)，可以使用-P 参数。这种方案的确可以提高性能，有很多使用 Redis 的应用在生产环境中这样做。

最后，基准测试需要使用相同的操作和数据来对比，如果这些不一样， 那

么基准测试是无意义的。

比如,Redis 和 memcached 可以在单线程模式下面对比 GET/SET 操作。两者都是内存数据库，协议也基本相同，甚至把多个请求合并为一条请求的方式也类似 （pipelining)。在使用相同数量的连接后，这个对比是很有意义的。

下面这个很不错例子是在 Redis（antirez）和 memcached（dormando）测试的。

antirez 1 - On Redis, Memcached, Speed, Benchmarks and The Toilet

dormando - Redis VS Memcached (slightly better bench)

antirez 2 - An update on the Memcached/Redis benchmark

你可以发现相同条件下面最终结果是两者差别不大。请注意最终测试时候，两者都经过了充分优化。

最后，当特别高性能的服务器在基准测试时候（比如 Redis、memcached 这类)， 很难让服务器性能充分发挥，通常情况下，客户端回事瓶颈限制而不是服务器端。 在这种情况下面，客户端（比如 benchmark 程序自身）需要优化，或者使用多实例， 从而能达到最大的吞吐量。

影响 Redis 性能的因素

有几个因素直接决定 Redis 的性能。它们能够改变基准测试的结果， 所以我们必须注意到它们。一般情况下，Redis 默认参数已经可以提供足够的性能，不需要调优。

网络带宽和延迟通常是最大短板。建议在基准测试之前使用 ping 来检查服务端到客户端的延迟。根据带宽，可以计算出最大吞吐量。 比如将 4 KB 的字符串塞入 Redis，吞吐量是 100000 q/s，那么实际需要 3.2 Gbits/s 的带宽，所以需要 10 GBits/s 网络连接， 1 Gbits/s 是不够的。 在很多线上服务中，Redis 吞吐会先被网络带宽限制住，而不是 CPU。 为了达到高吞吐量突破 TCP/IP 限制，最后采用 10 Gbits/s 的网卡， 或者多个 1 Gbits/s 网卡。

CPU 是另外一个重要的影响因素，由于是单线程模型，Redis 更喜欢大缓存快速 CPU， 而不是多核。这种场景下面，比较推荐 Intel CPU。AMD CPU 可能只有 Intel CPU 的一半性能（通过对 Nehalem EP/Westmere EP/Sandy 平台的对比)。 当其他条件相当时候，CPU 就成了 redis-benchmark 的限制因素。

在小对象存取时候，内存速度和带宽看上去不是很重要，但是对大对象（> 10 KB），它就变得重要起来。不过通常情况下面，倒不至于为了优化 Redis 而购买更高性能的内存模块。

Redis 在 VM 上会变慢。虚拟化对普通操作会有额外的消耗，Redis 对系统调用和网络终端不会有太多的 overhead。建议把 Redis 运行在物理机器上，特别是当你很在意延迟时候。在最先进的虚拟化设备（VMWare）上面，redis-benchmark 的测试结果比物理机器上慢了一倍，很多 CPU 时间被消费在系统调用和中断上面。

如果服务器和客户端都运行在同一个机器上面，那么 TCP/IP loopback 和 unix domain sockets 都可以使用。对 Linux 来说，使用 unix socket 可以比 TCP/IP loopback 快 50%。默认 redis-benchmark 是使用 TCP/IP loopback。

当大量使用 pipelining 时候，unix domain sockets 的优势就不那么明显了。

当使用网络连接时，并且以太网网数据包在 1500 bytes 以下时，将多条命令包装成 pipelining 可以大大提高效率。事实上，处理 10 bytes，100 bytes，1000 bytes 的请求时候，吞吐量是差不多的，详细可以见下图。

在多核 CPU 服务器上面，Redis 的性能还依赖 NUMA 配置和处理器绑定位置。最明显的影响是 redis-benchmark 会随机使用 CPU 内核。为了获得精准的结果，需要使用固定处理器工具（在 Linux 上可以使用 taskset 或 numactl）。最有效的办法是将客户端和服务端分离到两个不同的 CPU 来高校使用三级缓存。这里有一些使用 4 KB 数据 SET 的基准测试，针对三种 CPU（AMD Istanbul，Intel Nehalem EX，和 Intel Westmere）使用不同的配置。请注意，这不是针对 CPU 的测试。

在高配置下面，客户端的连接数也是一个重要的因素。得益于 epoll/kqueue，Redis 的事件循环具有相当可扩展性。Redis 已经在超过 60000 连接下面基准测试过，仍然可以维持 50000 q/s。一条经验法则是，30000 的连接数只有 100 连接的一半吞吐量。下面有一个关于连接数和吞吐量的测

在高配置下面，可以通过调优 NIC 来获得更高性能。最高性能在绑定 Rx/Tx 队列和 CPU 内核下面才能达到，还需要开启 RPS（网卡中断负载均衡）。更多信

息可以在 thread。Jumbo frames 还可以在大对象使用时候获得更高性能。

在不同平台下面，Redis 可以被编译成不同的内存分配方式（libc malloc, jemalloc, tcmalloc），他们在不同速度、连续和非连续片段下会有不一样的表现。如果你不是自己编译的 Redis,可以使用 INFO 命令来检查内存分配方式。请注意，大部分基准测试不会长时间运行来感知不同分配模式下面的差异， 只能通过生产环境下面的 Redis 实例来查看。

其他需要注意的点

任何基准测试的一个重要目标是获得可重现的结果,这样才能将此和其他测试进行对比。

一个好的实践是尽可能在隔离的硬件上面测试。如果没法实现，那就需要检测 benchmark 没有受其他服务器活动影响。

有些配置（桌面环境和笔记本，有些服务器也会）会使用可变的 CPU 分配策略。 这种策略可以在 OS 层面配置。有些 CPU 型号相对其他能更好的调整 CPU 负载。 为了达到可重现的测试结果，最好在做基准测试时候设定 CPU 到最高使用限制。

一个重要因素是配置尽可能大内存，千万不要使用 SWAP。注意 32 位和 64 位 Redis 有不同的内存限制。

如果你计划在基准测试时候使用 RDB 或 AOF，请注意不要让系统同时有其他 I/O 操作。避免将 RDB 或 AOF 文件放到 NAS 或 NFS 共享或其他依赖网络的存储设备上面（比如 Amazon EC2 上 的 EBS）。

将 Redis 日志级别设置到 warning 或者 notice。避免将日志放到远程文件系统。

避免使用检测工具，它们会影响基准测试结果。使用 INFO 来查看服务器状态没问题， 但是使用 MONITOR 将大大影响测试准确度。

不同云主机和物理机器上的基准测试结果

这些测试模拟了 50 客户端和 200w 请求。

使用了 Redis 2.6.14。

使用了 loopback 网卡。

key 的范围是 100 w。

同时测试了 有 pipelining 和没有的情况（16 条命令使用 pipelining）。

Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (with pipelining)

```
$ ./redis-benchmark -r 1000000 -n 2000000 -t get,set,lpush,lpop -P 16 -q
SET: 552028.75 requests per second
GET: 707463.75 requests per second
LPUSH: 767459.75 requests per second
LPOP: 770119.38 requests per second
```

Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (without pipelining)

```
$ ./redis-benchmark -r 1000000 -n 2000000 -t get,set,lpush,lpop -q
SET: 122556.53 requests per second
GET: 123601.76 requests per second
LPUSH: 136752.14 requests per second
LPOP: 132424.03 requests per second
```

Linode 2048 instance (with pipelining)

```
$ ./redis-benchmark -r 1000000 -n 2000000 -t get,set,lpush,lpop -q -P 16
SET: 195503.42 requests per second
GET: 250187.64 requests per second
LPUSH: 230547.55 requests per second
LPOP: 250815.16 requests per second
```

Linode 2048 instance (without pipelining)

```
$ ./redis-benchmark -r 1000000 -n 2000000 -t get,set,lpush,lpop -q
SET: 35001.75 requests per second
GET: 37481.26 requests per second
LPUSH: 36968.58 requests per second
LPOP: 35186.49 requests per second
```

更多使用 pipeline 的测试

```
$ redis-benchmark -n 100000
```

```
====== SET ======
  100007 requests completed in 0.88 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1
58.50% <= 0 milliseconds
99.17% <= 1 milliseconds
99.58% <= 2 milliseconds
99.85% <= 3 milliseconds
99.90% <= 6 milliseconds
100.00% <= 9 milliseconds
114293.71 requests per second
====== GET ======
  100000 requests completed in 1.23 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1
43.12% <= 0 milliseconds
96.82% <= 1 milliseconds
98.62% <= 2 milliseconds
100.00% <= 3 milliseconds
81234.77 requests per second
====== INCR ======
  100018 requests completed in 1.46 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1
32.32% <= 0 milliseconds
```

```
96.67% <= 1 milliseconds
99.14% <= 2 milliseconds
99.83% <= 3 milliseconds
99.88% <= 4 milliseconds
99.89% <= 5 milliseconds
99.96% <= 9 milliseconds
100.00% <= 18 milliseconds
68458.59 requests per second
====== LPUSH ======
  100004 requests completed in 1.14 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1
62.27% <= 0 milliseconds
99.74% <= 1 milliseconds
99.85% <= 2 milliseconds
99.86% <= 3 milliseconds
99.89% <= 5 milliseconds
99.93% <= 7 milliseconds
99.96% <= 9 milliseconds
100.00% <= 22 milliseconds
100.00% <= 208 milliseconds
88109.25 requests per second
====== LPOP ======
  100001 requests completed in 1.39 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1
54.83% <= 0 milliseconds
```

97.34% <= 1 milliseconds

99.95% <= 2 milliseconds

99.96% <= 3 milliseconds

99.96% <= 4 milliseconds

100.00% <= 9 milliseconds

100.00% <= 208 milliseconds

71994.96 requests per second

注意：包大小从 256 到 1024 或者 4096 bytes 不会改变结果的量级 （但是到 1024 bytes 后，GETs 操作会变慢）。同样的，50 到 256 客户端的测试结果相同。 10 个客户端时候，吞吐量会变小（译者按：总量到不了最大吞吐量）。

不同机器可以获的不一样的结果，下面是 Intel T5500 1.66 GHz 在 Linux 2.6 下面的结果：

$ ./redis-benchmark -q -n 100000

SET: 53684.38 requests per second

GET: 45497.73 requests per second

INCR: 39370.47 requests per second

LPUSH: 34803.41 requests per second

LPOP: 37367.20 requests per second

另外一个是 64 位 Xeon L5420 2.5 GHz 的结果：

$ ./redis-benchmark -q -n 100000

PING: 111731.84 requests per second

SET: 108114.59 requests per second

GET: 98717.67 requests per second

INCR: 95241.91 requests per second

LPUSH: 104712.05 requests per second

LPOP: 93722.59 requests per second

高性能硬件下面的基准测试

Redis2.4.2 。

默认连接数，数据包大小 256 bytes。

Linux 是 SLES10 SP3 2.6.16.60-0.54.5-smp，CPU 是 2 xIntel X5670 @ 2.93 GHz。

固定 CPU，但是使用不同 CPU 内核。

使用 unix domain socket：

```
$ numactl -C 6 ./redis-benchmark -q -n 100000 -s /tmp/redis.sock -d 256
PING (inline): 200803.22 requests per second
PING: 200803.22 requests per second
MSET (10 keys): 78064.01 requests per second
SET: 198412.69 requests per second
GET: 198019.80 requests per second
INCR: 200400.80 requests per second
LPUSH: 200000.00 requests per second
LPOP: 198019.80 requests per second
SADD: 203665.98 requests per second
SPOP: 200803.22 requests per second
LPUSH (again, in order to bench LRANGE): 200000.00 requests per second
LRANGE (first 100 elements): 42123.00 requests per second
LRANGE (first 300 elements): 15015.02 requests per second
LRANGE (first 450 elements): 10159.50 requests per second
LRANGE (first 600 elements): 7548.31 requests per second
```

使用 TCP loopback：

```
$ numactl -C 6 ./redis-benchmark -q -n 100000 -d 256
PING (inline): 145137.88 requests per second
PING: 144717.80 requests per second
MSET (10 keys): 65487.89 requests per second
SET: 142653.36 requests per second
GET: 142450.14 requests per second
INCR: 143061.52 requests per second
```

```
LPUSH: 144092.22 requests per second
LPOP: 142247.52 requests per second
SADD: 144717.80 requests per second
SPOP: 143678.17 requests per second
LPUSH (again, in order to bench LRANGE): 143061.52 requests per second
LRANGE (first 100 elements): 29577.05 requests per second
LRANGE (first 300 elements): 10431.88 requests per second
LRANGE (first 450 elements): 7010.66 requests per second
LRANGE (first 600 elements): 5296.61 requests per second
```

原文链接：http://www.udpwork.com/item/11757.html

TCP 网络协议及其思想的应用

大部分程序员都听说过 TCP/IP 网络协议，或者都写过 TCP socket 网络的程序，甚至还学过 TCP 原理，少部分看过 TCP 协议的某一个实现版本. 不过，真正掌握 TCP 原理及思想的人，我觉得不多. 只有理解了 TCP 原理及实现，并且把它背后的思想和技术活学活用到其它的领域，那才算是真正掌握了 TCP.

TCP 协议的目的，是在不可靠传输的 IP 层之上建立一套可靠传输的机制，它所应用的技术，如滑动窗口，慢启动，指数退避, Negal 算法等，都是优化目的.

活学活用 TCP，要活学活用它的哪部分呢？我觉得就是在不可靠传输之上建立可靠传输. 事实上，传输是一个广义的概念，不局限于狭义的网络传输，应该理解为通信和交互. 任何涉及到通信和交互的东西，都可以借鉴 TCP 的思想. 例如，当一个人向另一个挥手，另一个也要挥手回应，这样的场景是一种交互，也是一种通信.

可靠传输，最重要的是要区分出通信的两端，因为可靠传输不能脱离了通信的两端，一旦脱离了，可靠就会变成不可靠．例如，一些初学者使用 TCP socket 来开发聊天程序，但想当然的认为既然使用了可靠传输的 TCP 协议，那么整个系统就没有必要再考虑可靠传输的问题了．这是非常错误的，因为 TCP 的可靠只是对于它自身来说的，甚至是对于 socket 接口层，两个系统就不是可靠的了，因为 write() 发送出去的数据，没有确保对方真正的 read() 到．或者再往上一层，到了人机交互界面的一层，即使消息已经发送到了对方的聊天窗口，但对方可能没有看到(阅读到)，如果认为消息在聊天这一层层面已经可靠传输了，那就错了．

可靠传输的第一要素是什么？可靠传输的第一要素是确认，第二要素是重传，第三要素是顺序．任何一个可靠传输的系统，都必须包含这三个要素．

当然，数据校验也是必要的，但我们常使用的通信通道一般都提供了数据校验，所以经常依赖于底层，但要求特别高的场景，也需要在上层做数据校验，所以，我不愿意把数据校验也列为一个必要的要素．

所以，当你要设计一个可靠传输的系统，例如在 UDP 之上实现可靠传输，那么你就要立即想到这三个要素．当你要在任何一个可以进行通信的系统之上创建自己的可靠通信系统，无论这个底层的通信系统是以 API 的形式提供，或者是以服务的形式，只要它是一个通信系统，那么就可以，而且经常需要在上面创建自定义的可靠传输系统，那么你应该立即想到可靠传输的三要素．

原文链接：http://www.ideawu.net/blog/archives/782.html

【开源访谈】ECharts 作者 林峰 访谈实录

林峰，开源中国 @Kener-林峰，github @kener ，微博 @Kener-林峰，百度商业前端通用技术组，数据可视化方向负责人，资深前端研发工程师。喜欢设计，热爱编程，ZRender，ECharts 作者，目前专注于数据可视化方面的研究工作。

【软件简介】

ECharts，纯 Javascript 图表库，基于 Canvas，底层依赖 ZRender，商业产品通用图表库，提供直观，生动，可交互，可个性化定制的数据可视化图表，支持折线图（区域图）、柱状图（条状图）、散点图（气泡图）、K 线图、饼图（环形图）、雷达图（填充雷达图）、和弦图、力导向布局图、地图（内置世界地图、中国及全国 34 个省市自治区地理数据），同时支持任意维度的堆积和多图表混合展现。

【访谈实录】

1. 能否先介绍一下你自己（技术背景、工作经历、学习经历）？

没什么辉煌的经历，只是回想一下好像冥冥中安排好似的。

02 北邮计算机本科，算是科班生出身，计算机相关基础理论都是填鸭式的学了一遍，得益于各种实习经历那时觉得自己 C++写得不错，也得益于各种学生会活动积累了些 ps 的技能同时重新唤醒了自己从小到大对于设计兴趣，可以说从此坚定了自己未来职业规划方向是在编程+设计结合的领域。成绩凑合，06 年保研跟随杨放春和苏森教授研究方向是下一代网络， 依然写着 C++，MFC 嘛，揽了各种客户端用户界面的活，很花哨的为系统写换肤、个性化功能，回想起来还是挺有意思的。

毕业前一年机缘巧合遇到了一位资深顾问带着创业项目从美国回来，于是作为技术合伙人组建起技术团队，第一次接触 web，什么都不懂就开始没日没夜的封闭开发了半年，风风火火的在加州注册了公司，上线运营了半年发现不对劲。离毕业被赶出学校还有 2 周，我们散伙了，四散找工作去，还好这帮哥们都是比较 NB 的人物，没啥费劲就都有着落了，要不我真愧疚一生了。

这段经历基本改变了自己的编程领域，从 CS 到 BS，但并未改变方向，可以

说发现了一个更加吻合自己方向的职业：web 前端。于是投了百度，很幸运的被 Forain 师兄收留成了一名 FE，而且分配到一个对公司举足轻重的产品：凤巢系统。2 年多的摸爬滚打从菜鸟变成了高级菜鸟，成了凤巢前端的技术负责人，整天跟各种数据打交道，开始知道了数据的价值和力量，那是 2012 年末，大数据这名词才刚刚浮出水面，数据可视化更是（至少在国内）未被流传，乔帮主不让 i 系列上运行 flash 加上 html5 开始火热，我们需要寻求一个解决方案，用于凤巢系统数据报表的可视化展现，用于对凤巢系统用户体验监控数据的可视化展现等等，编程+设计+数据的结合的仿佛为自己量身定制，于是就转向了数据可视化的研究。百度前端领袖人物 Erik 回归后组建了商业前端的通用技术组，特意的规划出数据可视化方向，我也就顺理成章的从凤巢技术负责人的角色转到现在的角色，然后就挖了一个很深很深的坑（ECharts 的功能设计），紧接着的近 1 年的时间里就开始与团队一起一点点的填上这个坑。

2013 年 6 月 30 日，百度商业前端数据可视化团队带着 ECharts 1.0.0 与大家见面，半年多的时间，我们迭代发布了 10 个版本，成了“2013 年国产开源软件 10 大年度热门项目”之一，还在“2013 年度最新的 20 大热门开源软件”中排名第一，我很幸运代表团队被大家认识，十分感谢开源中国，感谢大家的支持。这就是今天的林峰。

2. 是什么促使你开发 ECharts ？

我想上一个介绍里已经基本回答了吧，对于个人，兴趣和职业规划导向，对于公司，强需求驱动。

3. 能否简单介绍一下 ECharts 以及它的应用场景？

数据可视化产品有很多，ECharts 定位在满足可复用的商业数据可视化需求，与业界已有的 Highcharts、FusionCharts 同级别的产品，当然他们都是成熟的商业收费产品，我们免费开源了，才起步，完善程度还有不小的差距。但不谦虚的说，ECharts 高度个性化和交互能力在不少方面已经成为了业界领先，拖拽重计算、大规模散点图获得了国家专利，数据视图、值域漫游、子地图模式也都是业界首创，独有的功能。至于应用场景就比较广了，不同行业都有各种需求。互联网就不用多说，报表系统、运维系统、网站展示，只要有数据展现的需求基本都能使用。像传媒，数据新闻在近几年也被越来越多的提及，财新网走得很前，他

们是最早使用 ECharts 作为数据新闻的可视化工具。各行各业其实都有营销展示、企业品牌宣传、运营收入的汇报分析等各种各样的应用场景，不管大数据是否被过渡热炒，数据确实已经成为很多企业最受重视的财富之一，有数据的地方基本都会有这方面的需求。

4. ECharts 目前是怎么推广的，在实际项目中的应用情况如何？

其实没啥推广，也就升级版本自己发发微博，在开源中国发个简单的升级的报道，没借助公司的力量，甚至内网新闻都没发过一篇。用口口相传可能不太合适，但确实就是靠大家相互传播的。

应用情况还好了，对于百度，毕竟这项目本身是为公司服务，整个商业体系的业务系统新需求开发基本都在使用 ECharts，原有业务的升级也都往这上迁移，目前 ECharts 不仅支撑起百度多个核心商业业务系统（如凤巢、广告管家、鸿媒体、一站式、百度推广开发者中心、知心业务系统等等）的数据可视化需求，还有为数众多的后台运维及监控系统（如百度站长平台、百度推广用户体验中心、指挥官、无线访问速度质量监控、凤巢代码质量统计报告等等）。

非百度使用 ECharts 情况我就不方便直接透露了，可以肯定的是比我们自己的项目要多，各行各业远超我们自己的预期。甚至还得到了跨领域以及国外技术团体关注，比如在 R 领域就同时出现国内外两个版本的接口扩展（其一就是 R 领域里大名鼎鼎的 Ramnath Vaidyanathan，ECharts 已被包含进 rCharts 的扩展中），其他开发语言还有 Julia 和 Python 接口扩展项目，这都是其他编程爱好者自发的项目。听说还有两家亚太地区金融咨询企业在研发基于 ECharts 的 BI 类产品，甚至还有人拿着 ECharts 跑到纽约市长数据分析部门的给他们展示，这都是我们的意外收获。再次感谢大家的支持。

5. 目前有哪些人参与到了 ECharts 的开发？平时花费多少时间精力在这个项目上？

目前参与开发的基本都是我们自己团队的人，并未有非百度的研发人员加入。具体是谁就不方便直接点名了，在我看来他们可都是技术 Genius，除了对这个方向有着极大热情之外大多还有着数学计算，图形图像处理方面的丰富经验，其他公司可别把他们挖走哦。其实也不是什么秘密，真有心通过微博或代码签名也都能把我团队成员全找到，ECharts 主页改版后我也争取把团队成员介绍给大家认

识。团队的核心成员包括我基本全力投向这个项目，除了升级维护还有项目支持的工作，其他的还有些游走在常规项目和可视化方向的同学参与着。

6. 你是否有通过 ECharts 获得收入？维护这个项目和你的全职工作如何平衡？

我很幸运，ECharts 项目本身就是为公司服务的，商业前端数据可视化方向的工作目前就是我的全职工作。收入是不会有的，不是没人给，找上门的外包项目、技术支持需求还是挺多的，只是项目性质以及公司规定我是不能从中获得收入的，包括很多公司邀请去分享介绍都提出能给费用的，我给的回答都是“免费能去，收费就不能去了”，作为开源项目，我们希望更多人使用，我们可以收到更多的反馈和建议以完善和推进这个项目，这对我们就是莫大的收益，所以我很乐意提供这方面的分享，并不以获得额外收入为目的。

7. ECharts 将来的发展方向？

技术发展很快，刚过去的这 3 个月我已经看到国外几大同类产品都发布了大升级，相关领域很多新创公司的成立，多起融资收购的报道，百度迁徙的出现也一夜间家喻户晓，可以说数据可视化的春天到了，很庆幸我们踩对了时代的步伐，但这本身也是挑战，新年回来我就跟团队同学说，只要我们慢下来，不出半年我们就会被遗忘。我定不出 3 年计划，3 年计划在互联网也是不适合的，甚至 1 年计划我都没完整写完，我只明确了这半年内 ECharts 要做什么，提高用户体验、性能、样式，让更多的人可以更加低成本的实现数据可视化需求。

8. 能否谈一下你对开源的理解，以及对国内开源技术和产品的看法？

在我看来，与人分享一个苹果，你只剩半个，但与人分享一个想法，你什么都没减少说不定还能收获更多的想法，这就是开源的魅力。可以肯定的说，如果 ECharts 没有开源，半年时间绝不可能有这个发展速度，半年多以来来自开源社区的需求和反馈从未间断，不仅如此，通过开源，我们还认识到国内外的领域专家、编程爱好者、社区、论坛各种公司，比如来自统计之都的现任秘书长 @cloud_wei 已经成为 ECharts 的推广大使，来自视觉中国的设计总监颜冬也在最近成了 ECharts 的首席设计师，还有像财新网 CTO @财财某、AdMaster 精硕科技 CEO @闫婴 AdMaster、@大数据文摘创办人汪德诚、@数据科学家联盟联合创始人兴宝、SupStat 联合创始人@陈堰平、@马金馨老师、@沐洲、@小雍子等等都

成了朋友，正是这些人和事极大的推动着 ECharts 的发展，代表团队感谢大家。

至于对国内开源技术和产品的看法，毕竟专精面向的领域还是比较窄的，我没啥发言权，好坏不论，延续开源的话题好了。确实看到国内开源氛围起来了，出现很多优秀的开源项目，不仅开发者个人，更重要的是公司企业的重视和支持，做得比较好的像阿里都已经有集团下的开源项目汇总了，而一直对开源并不十分积极的百度也在前几个月上线了 oss.baidu.com，像我所在的百度商业前端的通用技术组作为支撑整个百度商业体系产品的核心技术团队，我们的全部产出都是以开源方式发布，除了 ECharts，E 系还有像 ER（MVC 框架）、ESUI（商业 UI 库）、ESL（模块加载器）、ETPL（轻量级模板引擎）等数十个开源项目，更多的可以查看我们的 repository，第三方使用跟我们自己项目引入并没有任何差别，可以说这也是百度作为互联网领军企业之一对于开源社区的贡献和回馈。至于很多人都会有质疑说国内的很多开源项目都只是山寨国外的项目，像 ECharts 就很多人说山寨 Highcharts 的，我只能反问“有守门员就不让射球了？”良性竞争更会促进技术的发展，我希望大家更应该鼓励，人家先起步了，你现在走的每一步在其他人看来可能都是跟随着，但下一个弯道在哪？是否有超越的可能？甚至谁会跌倒谁会最先跑到终点又有谁敢定论？中国那么大，人才那么多，在任何一个技术领域成为全球领先都是可能的，这得靠大家一起的努力，如果你没参与进来，至少应该鼓励吧！

9. 你有什么建议给程序员初学者？

基础很重要，大牛经常给我灌输的一个观点，我看来也是这样的，就是 Sense，对技术的感觉，对技术的敏感度。编程基础都是通，像我以前也是写 C++，Java 的，转前端也就花了半年时间，不管你写的是什么，作为程序员我觉得应该有点“洁癖”，不是为实现而实现，想明白背后的原因，为啥我要这样写？有没有更好的做法？别人是怎么做到？为什么？不断的反复锻炼就是积累自己对技术的 Sense，你的算法能力、设计能力甚至架构能力也就慢慢上来了。

再一点就是一定一定要动手，看书 10 遍不及动手 1 遍，很多人问过我怎么学编程，我给的建议都是“找一本经典的书，读的同时从头到尾哪怕最简单的代码片段你都敲一遍、调通跑起来、然后自己完善做个升级，看完做完你就入门了

10. 能否给开源中国提一些建议？

开源中国做得很好了，没什么能说的。希望你们不要低调了，其实开源中国很多服务和产品都很棒，像 Git 代码托管、RunJs 在线编辑、讨论社区、城市圈活动都应该渗透进各种程序员的圈里，简单易用的加中文环境的工具其实对于很多初入行的程序员很有帮助，想当年 msdn 上提问那叫一个痛苦，时差问题半夜起来不说，码英文看英文也累。甚至有些可以作为服务提供出来，很多开源软件主页都是静态页面，如果可以把开源中国的讨论、在线编辑作为服务直接引入，那就大好了！

最后，感谢开源中国的帮助，你们在做着一件意义非凡的事，祝愿开源中国越办越好！

原文链接：http://www.oschina.net/question/947559_144622